CDL+CWS

UN LANGAGE DE PROTOTYPES ET SON ENVIRONNEMENT

par

David Jean

mémoire présenté au Département de mathématiques et d'informatique
en vue de l'obtention du grade de maître ès sciences (M. Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, décembre 1996

SOMMAIRE

Le but du projet était de créer un langage orienté objet (CDL, Cards Definition Language) pour le prototypage rapide d'applications appartenant à un même domaine. Celui qui nous intéressait était «les jeux de cartes solitaires». CWS, CardsWorkShop, représente l'environnement graphique multi-fenêtres de développement qui a été créé, formé d'un éditeur de texte contextuel, un compilateur CDL, un interprète de machine virtuelle et une bibliothèque d'objets et de codes prédéfinie.

Le premier chapitre décrit certains principes d'une facette de la programmation orientée objet : les prototypes. La place de CDL parmi les prototypes est expliquée à l'aide d'un exemple. La genèse de CDL et CWS est décrite, le tout suivi de la description du modèle CDL. Un second chapitre est dédié à la description du langage CDL. Nous verrons la syntaxe et la sémantique des éléments de base : constantes, variables, fonctions, objets et instances. Le troisième chapitre décrit l'implantation de l'environnement de développement, CWS, le compilateur, l'interprète et les objets prédéfinis.

REMERCIEMENTS

À Jacques Haguel pour sa supervision et sa confiance. Pour m'avoir donné la possibilité d'entreprendre un projet qui m'était cher et pour son don de montrer les évidences. Pour avoir rendu ce mémoire plus intéressant par ses idées et références. Finalement, pour toutes les fois qu'il m'a fait enlever le mot "toutes" avant un "les".

À Jean-Pierre Grenier qui influença remarquablement le projet. Je me souviendrai toujours de nos longues conversations.

À tous les utilisateurs Internet, spécialement Oliver Pretzel, qui nous ont encouragés ou aidés dans notre cheminement. À Shengrui Wang pour m'avoir introduit aux réseaux de neurones qui seront le départ de la deuxième génération du projet.

À mes juges, Michel Barbeau et Jean-Pierre Dussault, pour leurs commentaires constructifs et leur patience.

Finalement à mes parents, amis et collègues de travail qui m'ont poussé à finir mon mémoire. Merci.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

INTRODUCTION

CDL, Cards Definition Language, est le langage objet que nous avons créé pour le prototypage rapide d'applications appartenant à un même domaine. Un des buts particuliers était d'éviter l'obligation de créer une hiérarchie d'objets ou classes alors qu'une approche simple, par copie différentielle, ferait l'affaire. Il est spécialisé en développement de jeux de cartes solitaires. Au langage nous avons ajouté CWS, CardsWorkShop, un environnement graphique multi-fenêtres, formé d'un éditeur de texte contextuel, un compilateur, un interprète et une bibliothèque d'objets et de codes prédéfinie.

À l'origine un projet personnel, CDL et CWS sont devenus un projet de maîtrise. L'enthousiasme manifesté par des utilisateurs de divers coins du monde qui employaient le langage et l'environnement pour créer de nouveaux jeux de cartes ou autres, en réutilisant du code déjà écrit et en un temps assez bref, nous a incité à produire ce mémoire.

Dans le premier chapitre nous verrons certains principes d'une facette de la programmation orientée objet : les langages de prototypes. Dans le monde des prototypes, chaque objet ou instance peut servir à en créer un autre. Nous situerons CDL parmi les langages de prototypes à l'aide d'un exemple. La genèse du projet est ensuite

décrite. Le chapitre se termine par la description du modèle CDL, en particulier les objets et les instances, le recul d'opérations, les interfaces multiples et les itérations.

Un second chapitre est dédié à la description du langage CDL. Nous verrons la syntaxe et la sémantique des éléments de base et le pré-processeur. Nous verrons aussi la définition de constantes, variables et fonctions et leur utilisation dans les expressions. Nous décrirons les instructions, objets et instances et le processus d'instanciation.

Le troisième chapitre décrit l'implantation du compilateur et de l'interprète. Ces derniers sont fondés sur PL0 [Wir75]; nous verrons les modifications apportées pour analyser et exécuter notre langage. Nous finirons avec une description des objets prédéfinis.

Chapitre 1

Langages de prototypes et CDL

Dans ce chapitre nous verrons les prototypes et la place de CDL parmi ceux-ci. Nous verrons l'utilisation de CDL à l'aide d'un exemple de jeu. Nous poursuivrons par la genèse du projet, décrivant nos succès et déboires. Nous terminerons par la description du modèle CDL, en particulier les éléments le distinguant des autres langages.

1.1 Théorie des prototypes

Dans cette section nous discuterons d'une façon de représenter les connaissances dans le domaine objet : les prototypes. Dans leur livre sur les langages objets, Masini, Napoli, Colnet, Léonard et Tombre [Mas89] donnent la définition suivante :

prototype : individu moyen, considéré comme idéal, à partir duquel sont décrits les objets d'une même famille, dont il définit les propriétés caractéristiques. Les nouveaux objets sont produits par copie et modification des prototypes existants, plutôt que par instanciation. Ils ne contiennent que les propriétés différentielles de celles des prototypes dont ils sont la spécialisation.

À cette définition nous voudrions ajouter la notion de catégorie, ou famille, de prototypes. Même si dans un univers de prototype les nouveaux individus sont créés à partir d'un autre, il est impensable que n'importe quel individu puisse servir de point de départ à un autre. Notre définition est :

catégorie de prototype : classification qui limite les échanges constitutionnels aux membres de la catégorie (prototype ou objet). Il est possible de créer une nouvelle catégorie à partir d'une existante, mais les objets et prototypes de cette dernière ne pourront être créés à partir des prototypes de la première.

Pour reprendre l'exemple donné dans [Mas89], nous avons Clyde, l'éléphant qui nous est le plus familier, qui sera l'éléphant prototype, détenteur des propriétés et du comportement par défaut de tout éléphant. Le concept d'éléphant représente notre catégorie de prototypes. Les autres éléphants sont décrits par rapport à Clyde et en sont des spécialisations. La description d'un éléphant quelconque, appelons-le Bonnie, liée à celle de Clyde, ne contient que les comportements et caractéristiques qui diffèrent de ceux de Clyde. Ainsi, sauf indication contraire, le nombre de pattes ou de défenses, la couleur, etc. de Bonnie sont les mêmes que ceux de Clyde. Cependant, s'il s'avère que Bonnie est en réalité un éléphant rose, contrairement à Clyde, cette nouvelle couleur est simplement enregistrée dans la description de Bonnie, sans modifier le prototype. Bonnie peut devenir à son tour un prototype, celui d'éléphant rose.

1.1.1 CDL : un langage de prototypes

Ce n'est qu'après avoir fini le développement de CDL que nous nous sommes aperçu qu'il était un langage de prototypes. La plupart des concepts de CDL trouvent leur contrepartie dans les langages de prototypes.

En CDL, "objet"[1] signifie en fait "catégorie de prototypes". La notion d'objet en CDL sert à définir une catégorie de prototypes, c'est-à-dire une classification qui force les instances d'une catégorie à communiquer entre elles seulement. En plus de fournir un moyen de classification, l'objet fournit un prototype de base très simple. Dans l'exemple de l'éléphant, la catégorie éléphant, qui en CDL sera l'objet éléphant, devrait spécifier des éléments de base : possède quatre pattes, une trompe, une mémoire phénoménale, etc.

Deux niveaux de prototypes existent : instance virtuelle et instance. Ils permettent respectivement de définir des prototypes inanimés, représentant des sous-catégories de prototypes et des prototypes et individus animés.

Instance virtuelle est le niveau intermédiaire entre la catégorie et le prototype. C'est une spécialisation de la catégorie. Ce niveau n'a pas de contre-partie dans la théorie des prototypes, mais s'y insère très bien. L'instance virtuelle offre des prototypes inanimés pouvant être vus comme sous-catégories de prototypes. Pour poursuivre notre exemple,

[1] certains livres appellent les objets, comme nous les voyons, classes

une sous-catégorie de prototypes définie par l'instance virtuelle pourait être un éléphant africain. C'est un prototype de la catégorie éléphant possédant les particularités suivantes: oreilles et défenses gigantesques. Le prototype éléphant africain est une spécialisation de la catégorie éléphant, mais ne représente aucun individu en particulier. C'est un prototype inanimé.

La nuance entre instance virtuelle et héritage réside au niveau du comportement, ou liste de méthodes. Les instances virtuelles ne permettent pas d'en ajouter, alors que l'héritage le permet. Cependant, ils peuvent tous deux en redéfinir. Si une nouvelle catégorie de prototypes hérite d'une autre, cela complexifie la communication entre les prototypes, leur forme n'étant plus nécessairement la même. Nous devrions revoir notre restriction sur l'échange intra-catégorie du modèle.

L'instance est le prototype ou l'individu, créée par copie différentielle. Elle est une spécialisation de la catégorie (objet), de la sous-catégorie (instance virtuelle) ou d'un prototype (instance). La nuance entre prototype et individu est qu'un individu devient un prototype lorsque son comportement est assez intéressant pour qu'on décide de créer d'autres individus issus de lui. Si on regarde un graphe des liens de spécialisation, l'individu sera une feuille et le prototype sera un noeud. Les instances sont vivantes; elles peuvent envoyer et recevoir des messages et changer leurs comportements ou leurs caractéristiques.

Prenons par exemple un individu mutant Dumbo. C'est un éléphant africain, copie de son père Disné, lui-même un éléphant africain vivant dans un cirque. Son statut de mutant vient du fait qu'il possède une caractéristique des éléphants miquettes : il parle. De plus il a un comportement bien à lui : il vole. La figure 1 est le graphe de spécialisation de Dumbo.

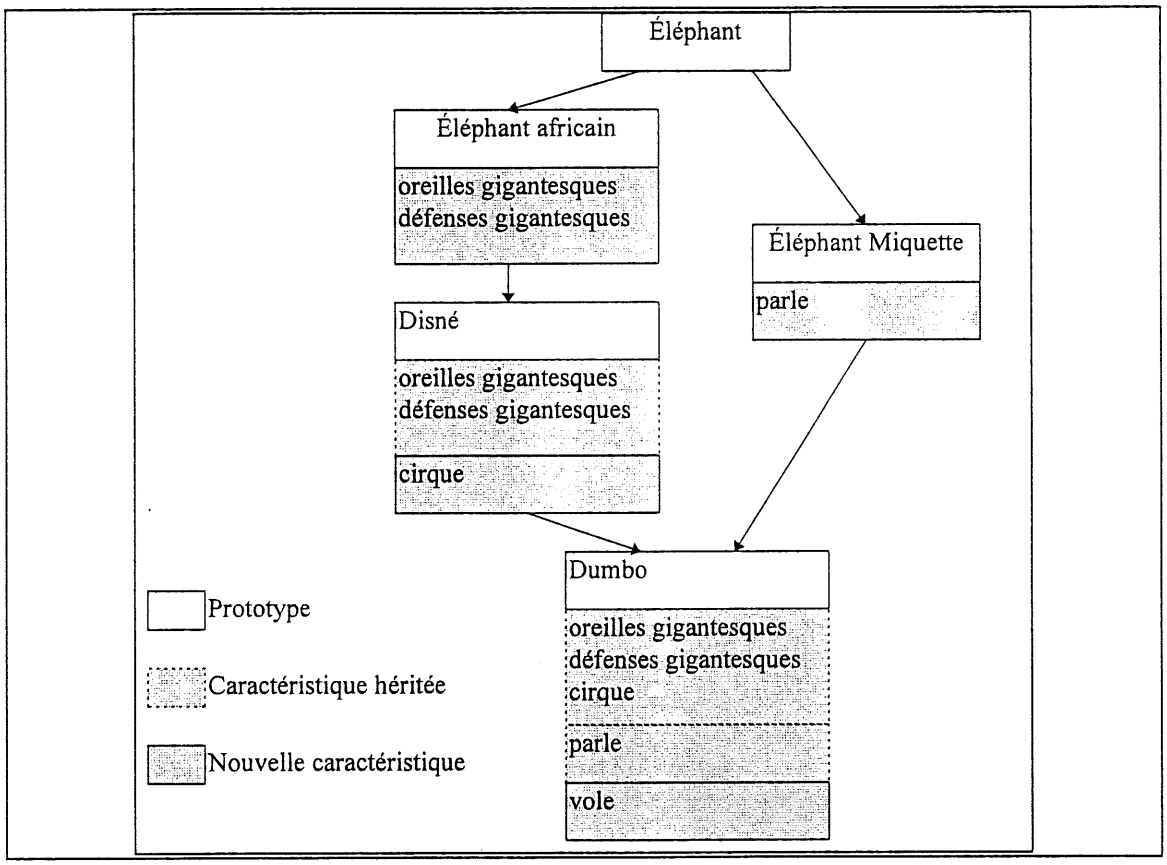


Figure 1 - Spécialisation de Dumbo

1.1.2 Terminologie

La présente section sert à clarifier les différents termes utilisés se rapportant au monde objet et à celui de CDL.

Objet

Un objet CDL se rapproche du concept de classe Java [Dav96] ou C++. C'est un groupe d'attributs et de méthodes servant à résoudre un problème. C'est un concept statique qui sert à générer des entités vivantes.

Instance

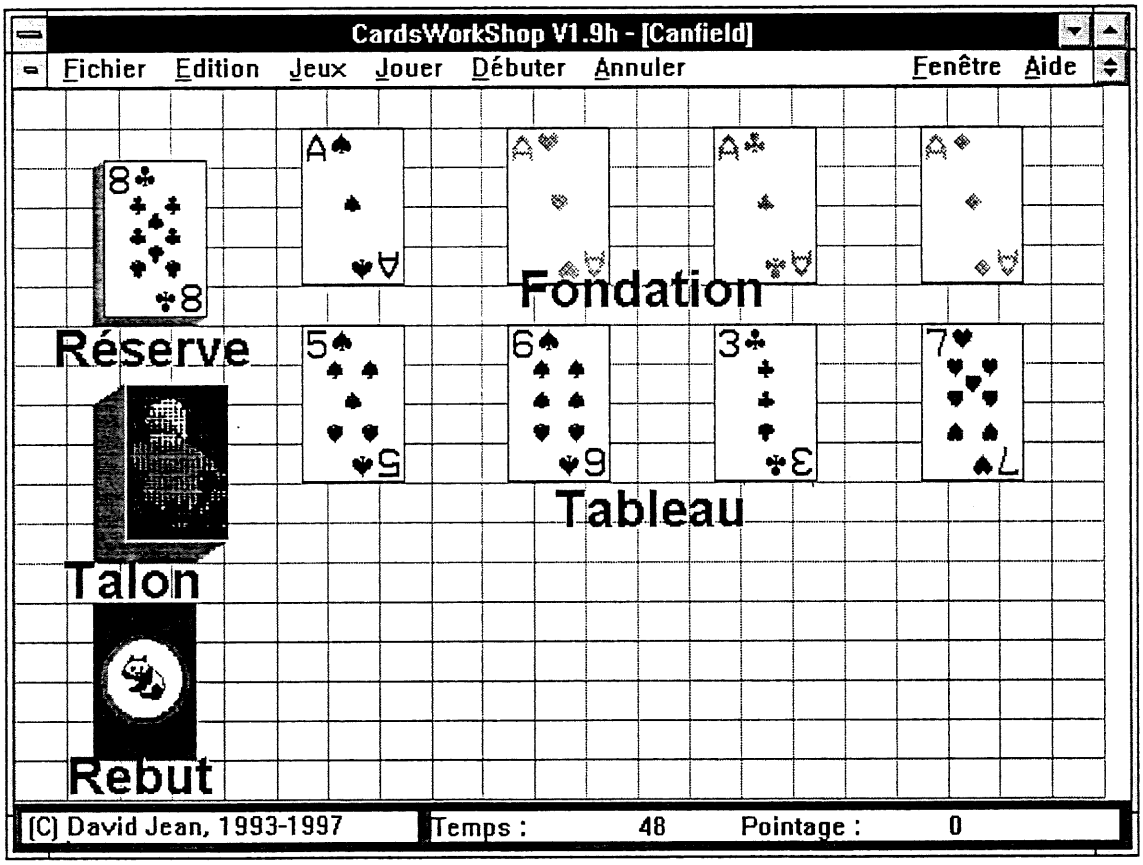
Une instance CDL se rapproche du concept d'objet Java ou C++. Dans ces langages, un objet est une instance d'une classe, alors qu'en CDL les entités vivantes sont des instances dérivées d'un objet (ou d'une autre instance).

Instance Virtuelle

L'instance virtuelle se rapproche d'une classe héritée de Java ou C++. C'est aussi un concept statique, mais en plus c'est un raffinement sur l'objet CDL (ou classe) qui a servi à la créer. Contrairement à une classe héritée, les champs d'une instance virtuelle possèdent une valeur. Les termes quasi-instance ou pseudo-instance auraient pu être utilisés.

1.1.3 Exemple de programme CDL

La première étape lorsqu'on entame un programme est d'identifier les éléments de la spécification. Comme exemple, nous avons choisi le jeu Canfield qui fut le point de départ par excellence pour les étapes du développement de CDL et CWS. Les règles, traduites de [Coo39], c'est-à-dire nos spécifications, sont les suivantes :



Distribution des cartes : battre un paquet de cartes et mettre treize cartes de côté dans une pile, face vers le bas. Tourner la carte au sommet afin qu'elle soit visible et placer la pile à gauche comme réserve. Prendre la prochaine carte du paquet et la placer sur la ligne de fondation : elle devient la carte de base de la fondation pour cette partie. Distribuer les quatre prochaines cartes du paquet sur le tableau.

But du jeu : essayer de libérer les cartes du tableau, réserve, rebut et talon afin de les monter en séquences ascendantes de la même suite sur la fondation.

Déplacement de cartes : les piles de fondations sont montées en séquences ascendantes de la même suite. Quand le roi est atteint, l'as, le deux, etc. le suivent jusqu'à ce que les treize cartes de la suite y soient. Les piles du tableau sont montées en séquences descendantes de couleurs alternées : rouge sur noir ou noir sur rouge. La carte au sommet de la réserve est disponible et jouable sur la fondation ou sur le tableau. Après que les déplacements possibles sur le jeu soient faits, trois cartes du talon sont transférées sur le rebut. Seule la carte au sommet du rebut est disponible. Une fois utilisée, la carte sous elle devient disponible et ainsi de suite. Les espaces vacants sur le tableau sont remplis en utilisant la carte au sommet de la réserve ou, si elle est vide, du talon ou du rebut. N'importe quelle séquence finale du tableau peut être déplacée ailleurs sur le tableau, tant que les règles de construction du tableau sont respectées. Une fois le talon vide, le rebut peut être réutilisé comme talon, et ce un nombre illimité de fois. Aucun nouveau battage n'est permis.

Des règles, nous déduisons un objet, ou catégorie de prototypes, pile (**Stack**). Cet objet, prédéfini, comporte des caractéristiques permettant de définir sa position, sa dimension et trois comportements :

- spécifier l'état initial (**`Start`**)
- essayer d'y prendre des cartes (**`SelectFrom`**)
- essayer d'y déposer des cartes (**`SelectTo`**)

Ces éléments ne font rien par défaut.

```
object Stack is
  const X : integer, Y : integer,
        W : integer, H : integer;
  procedure Start is doNothing;
  procedure SelectFrom(Spos : Index) is doNothing;
  procedure SelectTo(Spos : Index) is doNothing;
end Stack;
```

De cet objet nous pouvons spécialiser cinq types d'instances : fondation, tableau, réserve, talon et rebut. Nous allons définir une instance, virtuelle ou non, pour chaque type et elles seront nos prototypes. Souvent, ce qui fait la différence entre deux jeux de cartes, c'est le comportement des piles du tableau et de la fondation. Selon le jeu, nous devrions spécialiser le prototype de base de ceux-ci.

Une autre déduction provenant des règles est le nombre d'instances de chaque type. Dans Canfield, nous avons quatre fondations, quatre tableaux, une réserve, un talon et un rebut.

Les piles du tableau ayant le même comportement, nous n'aurons aucune spécialisation à ce niveau. La seule différence entre les instances est leur disposition sur la surface de jeu. Dans le but de réutilisation, nous créons une instance virtuelle pour un tableau. Ses caractéristiques sont :

- débute avec une carte tirée du talon
- accepte des cartes de couleurs alternées, de valeurs descendantes et permet de continuer avec le roi sur l'as
- permet le retrait illimité d'une séquence terminale de cartes par l'utilisateur.

```
virtual Stack tab_alt_wrap is
  Start is
    PULL 1 FROM talon;
  //***************************
  SelectFrom(Spos : Index) is
    PULL self! - Spos + 1 FROM self TO Cursor;
  //***************************
  SelectTo(Spos : Index) is
    IF self! = 0 THEN
      PULL Cursor! FROM Cursor TO self
    ELSE IF AlternateColor?(Cursor[1], self[self!]) and
            FollowRankWrap?(Cursor[1], self[self!]) THEN
      PULL Cursor! FROM Cursor TO self;
end tab_alt_wrap;
```

Pour faciliter la lecture du code en exemple, notons que :

- Les cartes en entrée et en sortie sont échangées avec une instance de pile nommée **`Cursor`** qui effectue le transport.
- Le symbole `!` est un opérateur post-fixé qui retourne le nombre d'éléments de la structure courante.
- Les symboles `[` et `]` forment un opérateur post-fixé qui retourne l'élément d'indice spécifié de la structure courante.
- L'identificateur **`self`** dénote l'instance courante.

Puis à partir de cette sous-catégorie de prototypes nous créons, par copie différentielle, quatre instances. Les seules caractéristiques redéfinies sont celles spécifiant la position et la dimension de la pile.

```
Stack tableau1 from tab_alt_wrap is
  X := 6;  Y := 7;
  w := 3;  h := 11;
end tableau1;

Stack tableau2 from tab_alt_wrap is
  X := 10;  Y := 7;
  w := 3;  h := 11;
end tableau2;

Stack tableau3 from tab_alt_wrap is
  X := 14;  Y := 7;
  w := 3;  h := 11;
end tableau3;

Stack tableau4 from tab_alt_wrap is
  X := 18;  Y := 7;
  w := 3;  h := 11;
end tableau4;
```

Nous procédons sensiblement de la même façon pour les fondations avec la particularité qu'une d'elles commence avec une carte. Nous créons une instance virtuelle pour une fondation qui accepte des cartes de la même suite, de valeurs ascendantes et qui permet de continuer avec l'as sur le roi. Aucun comportement de départ n'est défini.

```
virtual Stack found_same_wrap is
  SelectTo(Spos : Index) is
      IF SameSuit?(Cursor[1], self[self!]) and
         FollowRankWrap?(self[self!],Cursor[1]) THEN
        PULL 1 FROM Cursor TO self;
end found_same_wrap;
```

Puis à partir de celle-ci nous créons, par copie différentielle, une instance `foundation1` avec un comportement de départ (`Start`).

```
Stack foundation1 from found_same_wrap is
  X := 6;  Y := 2;
  w := 3;  h := 4;
  //**************************
  Start is
    PULL 1 FROM talon TO self;
end foundation1;
```

Les trois autres fondations proviennent directement du prototype de sous-catégorie et n'ont donc aucun comportement de départ.

```
Stack foundation2 from found_same_wrap is
  X := 10;  Y := 2;
  w := 3;  h := 4;
end foundation2;

Stack foundation3 from found_same_wrap is
  X := 14;  Y := 2;
  w := 3;  h := 4;
end foundation3;

Stack foundation4 from found_same_wrap is
  X := 18;  Y := 2;
  w := 3;  h := 4;
end foundation4;
```

Le talon est une pile avec deux comportements :

- débute avec un paquet de cinquante-deux cartes
- à la demande de l'utilisateur, transfère trois cartes sur le rebut ou, si le talon est vide, reprend les cartes du rebut.

```
Stack talon is
  X := 2;  Y := 9;
  w := 3;  h := 4;
  //**************************
  Start is
    ADD Ace+Spade..King+Diamond TO self;
  //**************************
  SelectFrom(Spos : Index) is
    IF self! = 0 THEN
      PULL rebut! FROM rebut TO self
    ELSE
      PULL 3 FROM self TO rebut;
end talon;
```

La réserve est une pile avec deux comportements :

- débute avec treize cartes tirées du talon
- permet de prendre sa carte au sommet.

```
Stack reserve is
  X := 2;  Y := 3;
  w := 3;  h := 4;
  //**************************
  Start is
    PULL 13 FROM talon TO self;
```

```
  //****************************
  SelectFrom(Spos : Index) is
    PULL 1 FROM self TO Cursor;
end reserve;
```

Le rebut montre une caractéristique intéressante : c'est une pile classique sans aucune spécification de départ mais qui a le même comportement que la réserve quant aux transactions de l'utilisateur. Nous créons donc une pile rebut et nous allons chercher le comportement de sélection source (**SelectFrom**) de la réserve :

```
Stack rebut is
  X := 2;  Y := 14;
  w := 3;  h := 4;
  //****************************
  SelectFrom from reserve;
end rebut;
```

Si nous modifions le comportement de sélection de la réserve, le rebut suivra automatiquement. Dans le cas de raffinement du code, c'est pratique; dans le cas de modification du comportement cela peut être dangereux.

Tous nos individus sont maintenant définis. Il nous reste à écrire deux prédicats essentiels: celui d'intégrité et celui de victoire. Le prédicat d'intégrité (**Integrity?**) est utilisé implicitement suite à une transaction de l'utilisateur pour s'assurer que l'état du système est stable. Il ressemble aux post-conditions d'Eiffel [Mey88] en ce sens que, si le test échoue, le système sera ramené jusqu'au dernier point stable. Nous pourrions écrire plusieurs règles pour vérifier l'état du système mais nous allons plutôt nous servir

de cette construction pour appliquer une règle du jeu que nous avons oubliée jusqu'ici : *les espaces vacants sur le tableau sont remplis en utilisant la carte au sommet de la réserve ou, si elle est vide, du talon ou du rebut.* Il serait complexe d'appliquer cette règle aux comportements du tableau, de la réserve et du rebut. Nous allons plutôt en forcer l'application automatiquement : nous vérifions chaque pile du tableau et si elle est vide nous allons chercher une carte de la réserve. du talon ou du rebut. Nous indiquons que le système est stable en retournant **TRUE**.

```
predicate Integrity? is
begin
  WITH it : Stack DO
    IF (it! = 0) THEN
      IF (reserve! > 0) THEN
        PULL 1 FROM reserve TO it
      ELSE IF (talon! > 0) THEN
        PULL 1 FROM talon TO it
      ELSE IF (rebut! > 0) THEN
        PULL 1 FROM rebut TO it
  FOR tableau1, tableau2, tableau3, tableau4;
  RETURN TRUE;
end;
```

Le prédicat de victoire est **Win?**. Il est utilisé à la suite d'une transaction pour déterminer une fin de partie victorieuse. Dans le cas de Canfield, cela arrive lorsque les fondations sont remplies :

```
predicate Win? is RETURN
   (foundation1! = 13) and (foundation2! = 13) and
   (foundation3! = 13) and (foundation4! = 13);
```

La dernière étape consiste à instancier les différents individus définis. L'instruction ORDER nous permet de spécifier ceux à créer et l'ordre dans lequel ils doivent l'être. Selon les règles du jeu, nous devrions commencer par créer le talon, la réserve et ensuite la première fondation. L'ordre des autres piles n'est pas important.

```
ORDER talon, reserve,
      foundation1, foundation2, foundation3, foundation4,
      tableau1, tableau2, tableau3, tableau4,
      rebut;
```

Cet exemple de programmation en CDL à été légèrement modifié pour simplifier la compréhension.

1.2 Genèse

La complexité d'un langage se propage à son compilateur. Le meilleur moyen d'arriver à ses fins est de commencer petit et d'augmenter l'envergure du projet au besoin et seulement lorsque le sous-ensemble précédent fonctionne [Wir75].

Débuter un projet n'est jamais facile, il faut déterminer les éléments d'avance et établir les liens entre ceux-ci. Notre approche fut de diviser le projet en étapes, chacune menant à un résultat tangible. La satisfaction de compléter une étape donna la motivation nécessaire pour attaquer la suivante.

La genèse de CDL et CWS se divise en deux phases. Durant la première le langage, le compilateur, l'interprète et un premier jeu, Canfield, allaient voir le jour. La seconde mena à la création d'un environnement complet, l'ajout d'une centaine de jeux et la distribution mondiale à travers le réseau Internet. Elle vit aussi l'évolution du langage jusqu'à son présent état, décrit dans cet ouvrage.

1.2.1 Phase 1 : noyau

Canfield, décrit à la section précédente, fut le point de départ du processus, divisé en six étapes :

Étape 1

Écriture d'une version non graphique (textuelle) de Canfield en PASCAL avec le concept objet simulé mais ressemblant déjà à la vision de CDL. Un seul objet existe : pile (**Stack**). Chaque instance peut être copiée à partir d'une autre ou chaque comportement peut individuellement provenir d'une autre pile. Le système est simulé à l'aide d'une fonction de distribution pour chaque instance et d'un "record" des champs variables. Une instance copiée partage la même fonction de distribution alors que si elle utilise une méthode d'une autre, une nouvelle fonction de distribution est créée et appelle cette autre méthode sur la réception du message. Un interprète de lignes de commandes simple permet à l'utilisateur de demander des requêtes pour le transfert de cartes d'une pile à une

autre. Les prédicats **`Win?`** et **`Lose?`** sont créés pour répondre au besoin de déterminer la fin de partie[2]. L'établissement de la règle de remplissage des entrées vides du tableau à l'aide de la réserve rend le code trop complexe. L'idée du prédicat **`Integrity?`** est alors amenée et permet de simplifier le code. La forme de celui-ci vient du principe de recul qui est éventuellement prévu. Pour le moment, la partie arrête si la non-intégrité du système est détectée.

Étape 2

Plutôt que de passer à l'écriture du langage, une bibliothèque graphique de bas niveau permettant de visualiser l'objet pile est implantée et ajoutée au code de Canfield. Celle-ci, écrite en PASCAL, deviendra la base du code externe nécessaire à la visualisation appelé à partir de CDL. Le concept de matrice virtuelle et l'instance de pile **`Cursor`** sont introduits pour permettre de déplacer visuellement les cartes; le code de Canfield est adapté en conséquence.

Étape 3

Écriture de la syntaxe et de la sémantique de la version 0.1 du langage CDL en BNF. Écriture des schémas de traduction des instructions. Écriture de Canfield.cdl avant même d'avoir un compilateur et un interprète. Décision sur le nom du langage : CDL pour Cards Definition Language.

[2] `lose?` est rarement implanté, sa logique étant compliquée

La définition d'objets y est absente mais les entrées correspondantes pour l'objet pile sont ajoutées à la table des symboles. En plus des fonctions et des procédures le concept simple de prédicat, une fonction booléenne, est ajouté. Une série d'instructions de manipulation de piles est définie (PULL, MOVE, TURN etc.).

Étape 4

Écriture du compilateur à partir d'une version modifiée du PL0 de Wirth. La décision d'utiliser l'interprète correspondant comme point de départ au nôtre permet de produire le code des nouvelles constructions au fur et à mesure. La spécialisation des champs et procédures au niveau des instances est ajoutée. Les nouvelles instructions sont implantées à partir des schémas de traduction.

Le concept d'objet contextuel est ajouté au contexte de procédure. Ceci permet de garder une référence à l'objet courant lors d'appels de procédures à d'autre méthodes d'instance ou d'objet. Une variable correspondante **`self`** est implicitement ajoutée à la table des symboles pour chaque méthode.

Étape 5

Écriture de l'interprète. Certaines nouvelles instructions machines sont implantées. En particulier une permettant d'appeler du code externe écrit en PASCAL (MSG). Le code graphique de la pile est intégré de cette façon. Une instruction d'échange de deux valeurs

(EXG) permet d'effectuer facilement le changement de contexte lors d'appels de méthodes.

Étape 6

Répétition du processus compiler-jouer-déverminage de Canfield. La première version est enfin née. Elle est simple et fragile mais fonctionnelle.

1.2.2 Phase 2 : évolution

Pour arriver à une version 1.0, la première "officielle", certains éléments furent ajoutés:

- Une instruction d'itération : WITH. Elle permet d'effectuer le même code sur plusieurs instances sans avoir à créer une méthode.
- Recul (Back-Tracking). Chaque instruction enregistre son inverse dans un registre. Par exemple, un PULL d'une pile à l'autre enregistrera le PULL de la deuxième à la première. Pour reculer d'une transaction il suffit d'exécuter le code correspondant dans le registre.
- Éditeur de texte intégré au compilateur et à l'interprète. Un bouton permet de compiler le code et les erreurs sont identifiées directement dans le texte.
- Une boite d'icônes, chacune associée à un fichier source ou exécutable. Permet rapidement d'exécuter un programme ou d'ouvrir du code dans l'éditeur.

- Un système d'aide. Une instruction WRITE est ajoutée. Elle ouvre une fenêtre de messages et y écrit le texte voulu. Une méthode Help est ajoutée aux piles et dans le mode d'aide est appelée au lieu de Select lors d'une sélection avec les boutons de la souris.
- Un fichier d'aide hypertexte décrivant le langage et l'environnement.
- Décision d'appeler l'environnement CardsWorkShop, l'atelier de cartes. Une erreur flagrante de marketing[3], plusieurs personnes n'y portant aucun intérêt, associant le nom à un programme de génération de cartes de souhaits. En plus les gens nous écrivant (en anglais) au sujet du logiciel l'écrivaient CardWorkShop (sans s), ce qui suggère une erreur d'anglais de notre part.

Cette version fut distribuée à travers Internet. Nous avons immédiatement commencé la définition et l'écriture de la version 1.2[4]. À partir de ce moment, Jean-Pierre Grenier devint un membre à part entière du projet. Son expérience acquise à programmer des jeux en CDL et ses connaissances du domaine objet amenèrent plusieurs améliorations au produit. Ce partenariat a abouti à la création d'une compagnie enregistrée pour la commercialisation de CardsWorkShop : SynHeme.

Entre les versions 1.0 et 1.2 nous avons reçu des commentaires et correspondu avec Quelques personnes sur Internet. Leur enthousiasme se faisait sentir.

---

[3] similaire à Chevrolet et sa voiture Nova, qui, en espagnol, veut dire : n'avance pas (No Va)

[4] certaines versions intermédiaires, comme 1.1, 1.3, 1.4, etc. ont été développées dans un but expérimental. Elles n'ont jamais été distribuées

Dans la version 1.2, en plus de corriger les erreurs de la précédente, nous avons ajouté certains éléments :

- Les icônes sont maintenant une représentation du jeu correspondant. L'environnement permet de prendre une image de la surface de jeu, la réduire et l'associer à celui-ci.
- L'éditeur amène le curseur au niveau du début de la ligne précédente sur un retour de chariot. Ceci accélère le développement en gérant une partie de l'indentation.
- Plus de jeux.

Après avoir rendu la version 1.2 disponible, nous nous sommes embarqué sur plusieurs modifications. Une version officielle 1.5 ne fut jamais distribuée faute de temps. Elle contenait les modifications et ajouts suivants :

- Début de découplage de l'environnement de jeu et de celui de développement en utilisant deux menus distincts.
- Un pré-processeur semblable à celui de C et la création d'une bibliothèque de codes dans des fichiers séparés incluables à la compilation. Jusqu'alors nous devions inclure le code à la main, ce qui compliquait le développement pour rien. Un fichier system.cdh fut créé. Il contient les définitions systémiques, constantes, objets et instances globales.
- Une instruction de délai : WAIT. Elle fut ajoutée pour permettre de chronométrer certains jeux.

- Un type GROUP permettant de grouper plusieurs instances d'un objet ou d'un type. L'instruction WITH fut modifiée pour accepter ce nouveau type. Celui-ci vient du besoin de Jean-Pierre qui voulait écrire du code générique pour traiter un ensemble d'instances de façon la plus abstraite possible, c'est-à-dire sans nécessairement savoir d'avance leur nombre ni leur identité. Comme la structure de tableau n'existe pas nous sommes arrivés à cette solution.
- Un objet **Cards**, qui rend possible l'accès à des images externes. Des jeux de tuiles à la Mah Jongh furent introduits sans rien ajouter d'interne au système.
- Plus de jeux.

Puis une version 1.7 fut planifiée. Elle contenait enfin des éléments qui auraient dû apparaître depuis le début :

- Création d'objets. N'importe quel objet peut maintenant être défini. Les entrées implicites ajoutées à la table des symboles pour les objets prédéfinis **Stack** et **Cards** sont transformées en définitions dans system.cdh
- Des méthodes d'objet sont ajoutées. N'importe quel prédicat, procédure ou fonction peut être préfixé par un nom d'objet et devient une méthode d'objet. Les références qui y sont faites vont être résolues dans le contexte de l'instance avant le contexte global.
- Plus de jeux.

La version 1.7 eut un plus grand succès et les commandes commencèrent à entrer de façon plus régulière (une ou deux par mois...). Finalement un utilisateur anglais, Oliver

Pretzel, nous envoya trois jeux qu'il programma à partir de règles transmises de génération en génération dans sa famille. Par contre la tendance semblait se dessiner : 80% des gens qui achetaient ne voulaient que jouer et non développer. Une décision de marketing fut prise par Jean-Pierre et moi : le système allait être divisé en deux produits, un environnement de développement et un de jeu. Seul celui de jeu serait disponible commercialement. Nous avions la frustration de savoir que certaines personnes programmaient leurs propres jeux et n'envoyaient pas d'argent. La version 2.0, appelée Cartivore dans son incarnation jeu, sortie au milieu de 1995, fut la dernière version officielle. Celle de développement la plus récente est 1.9h et garde le nom CardsWorkShop.

Les éléments ajoutés à cette dernière furent :

- Des objets fenêtre, bouton et port.
- La modification du module d'aide pour utiliser ces nouveaux objets.
- La table du pré-processeur fut enlevée et celle des symboles de compilation utilisée ce qui permet de compiler en fonction de constructions réelles. En particulier, ce changement provint du fait que lors de la programmation du module d'aide nous nous sommes aperçu qu'il était intéressant d'inclure automatiquement certains éléments si d'autres existaient. Par exemple si le programmeur définit une procédure **`About`**, la fenêtre d'aide possédera un bouton pour appeler celle-ci.
- Une instruction ON - DO, permettant d'exécuter du code dans le contexte d'une instance sans être obligé d'écrire une méthode spéciale.
- L'instruction WITH fut généralisée à n'importe quel objet ou type.

- Une méthode **`Register`** fut ajoutée par défaut aux objets[5]. L'instruction d'instanciation ORDER fut modifiée pour l'appeler.

- Une méthode **`Paint`** fut ajoutée aux objets. L'instruction DRAW, créée précédemment, l'appelle. Cette modification fut faite plutôt que d'éliminer DRAW pour garder une compatibilité avec les versions précédentes.

- Ajout d'instances virtuelles. Une longue discussion avec Jean-Pierre Grenier, résulta en cet ajout. Le besoin d'avoir des prototypes complets mais inactifs amena à cette décision. Une autre solution aurait été de considérer les instances inactives et de les rendre actives avec l'instruction ORDER. Un choix précédent qui émettait l'instruction de façon implicite pour les instances non explicitées força l'élimination de cette solution, encore une fois pour garder la compatibilité avec les programmes déjà existants.

- Une loi obligeant les fabricants de logiciels québécois à fournir une version en français de leurs produits nous força à rendre le nôtre bilingue. Le principe d'interface multiple fut très utile : les méthodes générant du texte, **`Update`** des boutons, ports et fenêtres et **`Help`** des piles, furent divisées en deux méthodes, anglaise et française. La méthode originale fut remplacée par une déclaration d'interface multiple. Les vieux programmes pouvaient fonctionner directement sans changement, générant le même texte dans le mode anglais ou français. Évidemment, nous dûmes les traduire quand même, mais sur une période de temps plus longue.

- Plus de jeux.

[5] **`Register`** est donc définie au niveau méta-objet

La version 2.0 fut distribuée sur Internet et nous avons demandé aux gérants de sites d'éliminer les versions 1.7 disponibles. Ces demandes furent sans effet et il existe encore plusieurs copies de la version 1.7 disponibles à ce jour sur des sites Internet. Il est évident que Cartivore n'ayant pas les possibilités de développement de son prédécesseur, plusieurs gérants préférèrent garder les deux versions.

Nous désirons arriver sur le marché avec une version plus évoluée de Cartivore en 1997. Nous espérons apprendre de nos erreurs et trouver une stratégie de marketing intéressante. Cependant la version 1.7 développée pour Windows 3.1 fonctionne à merveille sur Windows 95, ce qui va lui permettre de survivre encore longtemps.

1.3 Le modèle CDL

CDL est un modèle objet incluant des idées glanées à gauche et à droite, auxquelles furent ajoutées certaines idées originales. Le but, en créant CDL, était d'arriver à un langage permettant le prototypage rapide d'applications d'un même domaine, c'est-à-dire partageant des fonctions et manipulant des ensembles de données semblables. On pensera par exemple à des systèmes de paie, de gestion de pièces en industrie ou, comme dans le cas qui nous intéresse, des jeux de solitaire. Le tableau 1 donne des exemples pour chacun.

La rapidité de prototypage de CDL provient de sa syntaxe simple, d'instructions de base puissantes, de quelques objets de base pré-écrits et d'une bibliothèque de codes ou de comportements réutilisables. Sa syntaxe est voisine de PASCAL et d'Eiffel, c'est-à-dire avec plusieurs mots clefs descriptifs au lieu de symboles comme C++. Une des instructions de base qui donne sa puissance à CDL est l'itération. Elle permet d'effectuer un traitement sur une collection d'instances.

Type de système	Données manipulées	Fonctions partagées
paie	employé adresse salaire	ajouter(employé) calculer(salaire)
gestion de pièces	numéro de pièce fournisseur	commander(pièce) pièce_disponible?(pièce)
jeux de solitaires	carte pile	empiler(carte,pile) dépiler(pile)

Tableau 1 - Exemple de systèmes, données et fonctions

Une des erreurs de Wirth lors de la création du PASCAL fut l'absence d'une bibliothèque prédéfinie de codes de base. Chaque implanteur fut dès lors obligé d'écrire la sienne, ce qui rendit les programmes PASCAL non-portables. Cette erreur fut évitée pour C, surtout grâce à des organismes internationaux de standardisation comme ANSI. Wirth corrigea son approche lors de la création du langage Modula.

Voici les caractéristiques principales de CDL qui seront décrites en détail dans les sections qui suivent :

- Deux niveaux de définition normale, objet et instance (1.3.1)

- Deux niveaux de définition intermédiaire, méta-objet et instance virtuelle (1.3.1)
- Possibilité d'héritage simple (1.3.2)
- Partage de fonctionnalité entre instances (1.3.2)
- Partage de fonctionnalité entre méthodes d'une même instance (1.3.2)
- Interfaces multiples (1.3.2)
- Post-conditions, permettant le recul automatique de transactions[6] (1.3.3)
- "log" permettant le recul des transactions accumulées (1.3.3)
- Paquetage partitionnable (1.3.4)
- Constantes définissables par instance (1.3.4)
- Itérateur (1.3.5)

1.3.1 Objets, instances virtuelles et instances

CDL offre deux niveaux de définition normale : objet et instance et deux niveaux intermédiaires : méta-objet et instance virtuelle. La hiérarchie est décrite figure 2.

Le niveau méta-objet, comme défini par Cointe [Coi87], décrit les mécanismes implicites à chaque objet. Dans notre cas, un langage compilé, ce niveau permet de définir le fonctionnement interne qui rend possible l'interaction d'instances au moment de

[6] à la Eiffel

l'exécution, c'est-à-dire le passage de messages, la création tangible des instances et leur destruction.

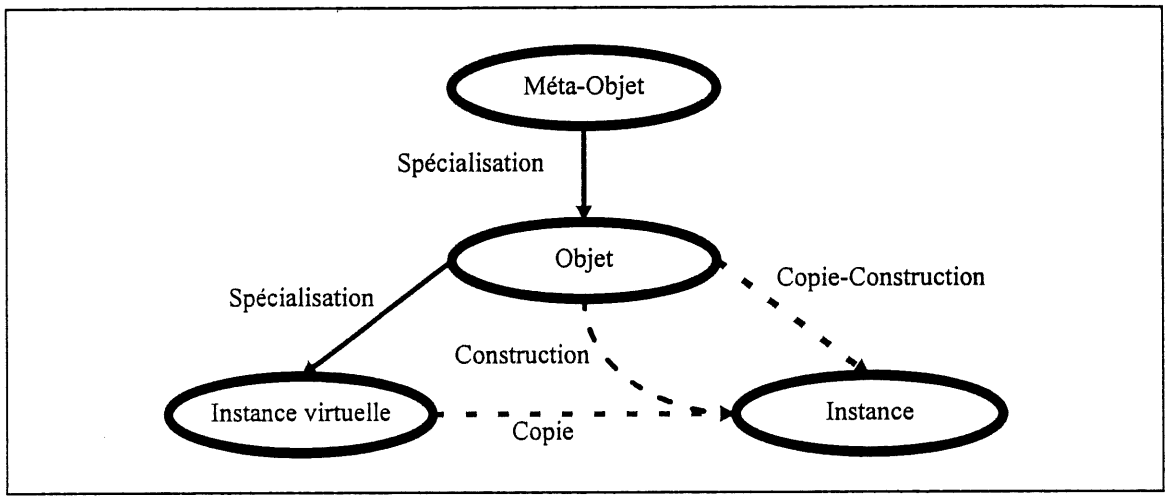


Figure 2 - Hiérarchie objet de CDL

Les trois niveaux suivants sont ceux qui nous intéressent vraiment : objets, instances virtuelles et instances. L'objet définit les caractéristiques communes aux instances ou instances virtuelles créées à partir de son modèle : les attributs (variables et constantes) et comportements (méthodes) et les valeurs par défaut de ces caractéristiques. Nous voyons ici la première différence avec les modèles objets usuels : le niveau objet offre un moule et un comportement de base mais les niveaux inférieurs, sans utiliser le concept d'héritage, auront la liberté de changer leur comportement au moment de la définition. C'est le clonage différentiel comme défini dans les langages à prototypes [Mas89].

L'objet offre un moule et des caractéristiques de base alors que l'instance virtuelle n'offre que des caractéristiques redéfinies. Elle est définissable de la même façon qu'une

instance normale mais avec la particularité de ne jamais devenir une structure tangible au moment de l'exécution.

Les instances virtuelles étaient absentes de la première version de CDL. Elles furent ajoutées lorsque nous avons réalisé que nous nous dirigions vers la même erreur que Wirth, où chaque utilisateur écrit sa propre version des comportements communs. Nous avons donc ajouté les instances virtuelles et procédé à l'écriture de plusieurs de celles-ci pour les comportements standard des jeux de cartes.

L'instance, la forme la plus simple, est la seule à exister de façon tangible. Une instance naît en prenant de l'espace mémoire, envoie et répond à des messages durant sa vie, puis meurt, libérant l'espace mémoire qu'elle utilisait. L'instance peut être construite sur le modèle de base de son objet ou en copiant une instance virtuelle, elle-même construite à partir de l'objet de base. Le comportement individuel des instances CDL pourra être modifié préalablement à sa création, une caractéristique des langages de prototypes.

Pour faire une comparaison avec le monde réel, on peut voir l'objet comme un plan décrivant une personne et son habillement, par exemple un gilet et un pantalon. On peut ensuite voir les instances virtuelles comme des mannequins en cire offrant une tenue vestimentaire prêt-à-porter, gilet polo rouge et pantalon de denim kaki, T-Shirt noir avec boxer rayé rouge, etc. L'instance sera la personne en chair et en os qui décidera comment s'habiller en copiant le linge porté par un mannequin. Le mannequin est proche de la personne, mais il ne prendra jamais vie. Dans cet exemple, décrire le rôle du méta-objet,

c'est-à-dire quels sont les mécanismes qui donnent la vie, relèverait de la théologie ou de l'évolutionnisme...

1.3.2 Héritage, partage de fonctionnalités et instances virtuelles

CDL a été développé avec la possibilité d'héritage simple en tête. Les caractéristiques techniques de cet héritage simple existent sur papier mais n'ont jamais été implantées. Cela vient du fait que la possibilité du partage de fonctionnalités entre instances a, dans notre cas, supprimé le besoin d'avoir un mécanisme d'héritage implanté au niveau supérieur, c'est-à-dire celui de l'objet. Après plus de soixante-quinze applications bâties avec CDL et CWS, nous continuons à croire en notre décision.

En fait, le partage de fonctionnalité entre instances et le concept d'instances virtuelles donnent une partie de la puissance de l'héritage multiple, c'est-à-dire le regroupement de comportements de provenances différentes dans un même ensemble. Nous pouvons créer une instance rapidement en allant chercher directement des éléments à gauche et à droite dans d'autres instances ou instances virtuelles, en plus de pouvoir conserver les comportements par défaut de l'objet. Dans certains autres langages objets observés, on devra créer un autre (ou d'autres) objet héritant des objets sources puis construire une instance; tout cela, parfois, pour une seule instance.

En plus du partage entre instances, plusieurs méthodes de la même instance peuvent partager le même comportement. Dans le cas de CWS par exemple, l'objet **Stack** a des méthodes pour le dépôt et la prise de cartes avec le bouton gauche ou droit de la souris. Cela représente quatre méthodes : **`SelectLeftFrom`**, **`SelectLeftTo`**, **`SelectRightFrom`**, **`SelectRightTo`**. De nouvelles interfaces multiples ont été ajoutées à l'objet **Stack**, permettant de définir le comportement de plus d'une méthode à la fois. La figure 3 décrit les méthodes multiples de l'objet **Stack**; ce sont celles qui ne sont pas encerclées. Par exemple, pour définir le comportement d'une pile lorsque n'importe quel bouton de la souris est utilisé pour prendre des cartes, on définira **`SelectFrom`**, ce qui affectera le comportement défini aux méthodes **`SelectLeftFrom`** et **`SelectRightFrom`**. La pile peut être vue à travers deux interfaces mutuellement exclusives :

```
Stack A1 is
  SelectLeftFrom(SPos : index) is PULL 1 FROM B1;
  SelectRightFrom(SPos : index) is PULL 1 FROM B1;
end A1;
```

ou

```
Stack A1 is
  SelectFrom(SPos : index) is PULL 1 FROM B1;
end A1;
```

Évidemment, les méthodes de base qui partageront le même comportement doivent avoir des listes de paramètres identiques.

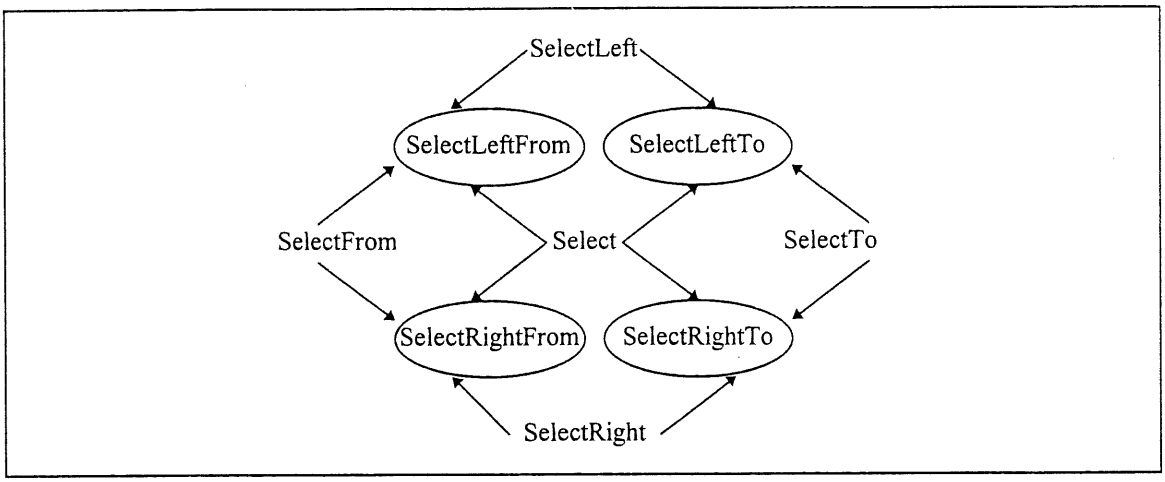


Figure 3 - Méthodes multiples de l'objet Stack

1.3.3 Post-conditions et recul

Il existe un prédicat d'intégrité global qui est vérifié après chaque transaction et qui peut déclencher son retrait. Une transaction commence par une pression sur un bouton de souris et se termine par son relâchement. L'algorithme est décrit section 3.3.1.

Cette post-condition peut servir à détecter et à protéger le système de certaines erreurs du programmeur. C'était le but original. Par contre comme c'est le programmeur lui-même qui est responsable d'écrire cette condition, les problèmes peuvent persister. Il aurait fallu, comme Meyer en Eiffel [Mey88], appliquer une post-condition par instance et non de façon globale.

Peu à peu le prédicat est devenu une façon de forcer l'application de certaines règles, comme le montre l'exemple de Canfield, section 1.1.3. Le prochain exemple, adapté du jeu pyramide, montre l'utilisation du prédicat d'intégrité à sa vraie fin. Dans ce jeu, il faut sélectionner dans une pyramide une ou deux cartes qui totalisent treize. Si plus de deux cartes sont sélectionnées ou si le total des deux cartes est différent de treize, la dernière transaction est refusée :

```
predicate Integrity? is
var nb_select, total_select : integer;
begin
  nb_select := GetNumberOfSelectedCards(tableau);
  total_select := GetTotalOfSelectedCards(tableau);

  IF nb_select > 2 THEN
    RETURN FALSE
  ELSE IF total_select = 13 THEN
    begin
    RemoveSelectedCards(tableau);
    score:=score+2;
    RETURN TRUE;
    end
  ELSE IF nb_select = 2 THEN
    RETURN FALSE
  ELSE
    RETURN TRUE;
end;
```

En plus du prédicat d'intégrité, une fonction **Undo** permet de reculer. Elle est accessible depuis le menu de l'environnement.

Le registre permettant le recul des transactions accumulées peut être vu comme une mémoire de code. Au début d'une transaction, une instruction de fin est ajoutée. Puis,

chaque modification à une variable globale ou d'instance (les éléments des piles inclus) enregistre l'action contraire. Dans le cas d'une affectation, l'action contraire est aussi une affectation mais avec la valeur précédente. Pour reculer, le code est exécuté et enlevé en ordre inverse jusqu'à l'instruction de fin. Par exemple, si avant l'exécution de **SelectFrom** de l'instance **tableau1** la variable **a** vaut 6,

```
Stack tableau1 is
  SelectFrom(Spos : index) is
    begin
    a := a + 10;
    PULL 1 FROM self TO foundation;
    end;
end tableau1;
```

le code enregistré ressemblera à ceci :

```
RETURN;
a := 6;
PULL 1 FROM foundation TO tableau1;
```

Comme **a** valait 6 avant l'exécution, c'est une affectation de 6 à **a** qui est enregistrée. Les références dynamiques, comme **self**, sont remplacées par l'instance, celle-ci étant connue à ce moment. Il suffit d'exécuter ce code en ordre inverse pour ramener le système à son état précédent. Évidemment, ce processus étant effectué au niveau de l'interprète, le code enregistré est sémantiquement semblable à cet exemple mais sous une forme binaire.

1.3.4 Paquetage et définition des objets et instances

Le contenu de chaque objet et instance est défini dans une construction associée. Ce paquetage est partitionnable, c'est-à-dire qu'il peut être fractionné et réparti dans le code. L'objet fenêtre illustre l'utilité de ceci :

```
object Window is
  const parent : Window,
        X : integer := 0, Y : integer := 0,
        ...
        text : string;
  var modified : boolean := FALSE;
  procedure Init;
  procedure Register is begin asmx self; asm MSG 2 0; end;
end Window;

Window MAINWINDOW is
  parent := nil;
  X := 0; Y := 0;
  ...
  Register is begin asmx self; asm MSG 2 1; end;
end MAINWINDOW;

object Window is
  const parent : Window := MainWindow;
end Window;
```

Le parent d'une fenêtre en est une autre et nous savons d'avance que dans la majorité des cas ce parent sera la fenêtre principale (**MainWindow**). Nous ne pouvons utiliser cette dernière sans l'avoir définie et nous avons besoin de savoir ce qu'elle est pour la définir. Le paquetage partitionnable nous permet de définir l'objet fenêtre avec un parent indéfini,

de décrire l'instance **`MainWindow`** et de changer sa valeur par défaut au niveau de l'objet.

Dans chaque partition subséquente à la première les caractéristiques définies précédemment, méthode ou constante, peuvent être valuées ou de nouvelles peuvent être ajoutées. Aucun retrait n'est possible ni changement de type afin de ne pas affecter des constructions définies entre les partitions.

Constantes et dynamisme

Les constantes définies et valuées au niveau de l'objet peuvent se voir attribuer une nouvelle valeur lors de la définition d'une instance, de même que lors d'une copie de prototype. Leur statut n'est donc pas constant de façon statique mais le devient au niveau dynamique.

Comme à l'exemple précédent, l'instance **`MainWindow`** change la valeur des constantes **`parent`**, **`X`** et **`Y`** au moment de la compilation. Ces mêmes attributs sont cependant constants et non-modifiables à l'intérieur des méthodes. L'état de construction du système est complètement déterminé à la compilation.

En CDL, les opérations de création aux niveaux des objets et des instances sont statiques. Aucun mécanisme dynamique d'instanciation n'existe. C'est pourquoi CDL offre

plusieurs facilités de définition. Le principe qui consiste à changer la valeur des constantes par instance est l'équivalent des paramètres du constructeur des langages objets.

Ce manque de dynamisme fonctionne bien pour de petits systèmes manipulant un nombre fini de données, comme les cinquante deux cartes d'un paquet. Dès qu'un doute sur la quantité de données survient, le modèle statique de CDL devient limitatif. Le langage n'empêche pas l'ajout de constructions dynamiques mais aucun développement en ce sens n'a été effectué à ce jour.

1.3.5 Itérateur d'instances

Deux caractéristiques de CDL : une structure typée de liste d'instances (GROUP) et une fonction d'itération (WITH), simplifient une partie habituellement lourde d'un programme : les itérations. Ces listes, structures de base connues du compilateur CDL, permettent de grouper plusieurs instances et de rendre les manipulations sur celles-ci plus faciles car le transport des instances d'une fonction à l'autre s'effectue à l'aide d'un seul identificateur, celui de la liste, au lieu d'un par instance. Là où la puissance ressort, c'est au niveau de l'instruction d'itération qui, en plus de pouvoir itérer sur une série d'instances nommées littéralement, peut itérer sur chaque élément de la liste.

Dans une instruction d'itération, on commence par donner un nom à l'instance utilisée dans le corps de l'itération. À chaque itération il y aura un lien dynamique entre ce nom et l'instance courante. Il est à noter que tout type de base est utilisable dans une instruction d'itération. Cette particularité augmente le modèle objet à ce niveau en traitant les types de bases comme des objets, un concept parfois manquant dans le monde objet des langages compilés. Java adhère à ce principe à l'aide de classes prédéfinies qui correspondent à chaque type de base [Dav96], comme par exemple les classes Integer et String. C++ n'offre rien à ce niveau. À l'exemple suivant, le nom utilisé dans l'itération est `I`.

On définit ensuite le corps de l'itération puis on liste les instances dans l'ordre où elles seront traitées. L'exemple suivant montre deux façons de définir les listes d'instances :

```
l := 0;
WITH I : Stack DO
     l := l + I.lonely
FOR inst1, inst2, inst3;


const group1 := | inst1, inst2, inst3 |;

l := 0;
WITH I : Stack DO
     l := l + I.lonely
FOR group1;
```

Dans le premier cas on liste littéralement les instances sur lesquelles on veut itérer alors que dans le deuxième cas on donne seulement une constante de liste. Le code écrit dans

le deuxième cas sera visiblement plus facile à réutiliser. Si on l'inclut dans une fonction et qu'on passe cette liste en paramètre, ce bout de code sera écrit définitivement.

Une vérification statique sera effectuée pour s'assurer que les instances à itérer sont du même type que le littéral défini dans l'en-tête de l'instruction. La vérification étant statique, il reste la possibilité qu'une liste d'instances du mauvais type soit liée dynamiquement à la variable de liste. Ce problème pourrait être résolu en étendant le type des variables de liste par un sous-type décrivant le type commun des instances de la liste. Si on souhaite avoir des listes hétérogènes, un mécanisme dynamique de typage devra être introduit. Évidemment l'instruction d'itération, dans son état actuel, ne permet pas les listes hétérogènes.

Chapitre 2

Le Langage CDL

Ce chapitre décrit la syntaxe et la sémantique de CDL. Dans la première partie, nous décrirons la structure des programmes, les éléments syntaxiques de base et le pré-processeur. Ensuite nous verrons comment définir des variables et constantes et comment les utiliser dans des expressions. Puis deux sections décriront la définition d'objets et celle d'instances et leur création. Finalement, dans la dernière section, nous traiterons de la définition de fonctions et des instructions de base de CDL.

2.1 Structure d'un programme CDL

Un programme CDL est un fichier divisé en sections. Il est aussi possible de diviser le code en plusieurs fichiers et de les inclure un à un dans le fichier principal à l'aide du pré-processeur. Les fichiers composant la bibliothèque CDL sont faits ainsi, prêts à être inclus dans le fichier principal de chaque application. La figure 4 décrit la structure d'un programme CDL en format BNF.

La clause ***header*** indique deux informations, le nom du programme exécutable produit par la compilation et la dimension de la matrice virtuelle de la fenêtre principale. La

partie de définition de la matrice aurait dû être éliminée lors de l'introduction de l'objet fenêtre car il y a ambiguïté entre elle et l'instance prédéfinie MainWindow. La décision de laisser cette clause vient du fait qu'une première version de CWS avait déjà été distribuée sur Internet lorsque les nouveaux objets, comme fenêtre, furent introduits. Nous voulions garder une compatibilité entre les versions.

```
program ::=  header ';'
             #include 'system.cdh'
             (object_def | instance_def |
             const_def | var_def |
             pred_def | proc_def | func_def |
             order_def )*
             [ DO statement ]
             #include 'main.cdh'
             '.'

header ::=   GAME id IS integer BY integer
```

Figure 4 - BNF d'un programme CDL

Le coeur du programme est divisé en sections qui peuvent être répétées ou être absentes. Elles permettent de définir des objets, des instances, des constantes, des variables et des fonctions. Elles sont décrites dans les sections suivantes.

L'instruction **DO *statement*** suivant les sections de définitions est exécutée une fois, lorsque le programme débute, comme le bloc **BEGIN *statement* END** global d'un programme PASCAL. Jugé inutile au début et brisant l'idéologie objet par son caractère procédural, ce bloc fut ajouté pour répondre à certains besoins, par exemple le cas où

certaines configurations globales modifiables par l'utilisateur doivent être transportées d'exécution en ré-exécution.

Les deux instructions de pré-processeur **#include** soulignées dans la figure 4 sont implicites. Elles permettent de définir des constantes, variables, procédures et objets globaux. Ceci force chaque installation du langage à être dédiée à un type d'application[7]; par contre cela enlève quelques secondes au processus de développement en fournissant l'environnement de base dans n'importe quel fichier de code source vide, contrairement aux autres langages comme C ou C++ ou certains fichiers sont inclus à tous coups[8].

2.1.1 Unités lexicales

CDL est un langage à mots clefs, contrairement à certains langages comme C ou C++ qui sont des langages à symboles clefs. Là où en C on écrira &&, en CDL ou en PASCAL on écrira le mot **and**. Le tableau 2 décrit les unités lexicales, leur sens et quelles règles on doit respecter pour les utiliser ou en créer de nouvelles.

[7] correspondant aux définitions dans `systems.cdh` et `main.cdh`

[8] par exemple : `stdio.h` ou `stream.hpp`, ou dans le cas de Microsoft Windows : `windows.h`

Unité lexicale	Littéral	Sens	Règles de construction/utilisation
id		Sert comme référence pour une valeur, une fonction, un objet ou une instance.	Peut être construit à l'aide de combinaison des caractères alphanumériques et des caractères _ et ?. Peut débuter par un chiffre et doit être différent des mots clefs.
Entier		Une valeur entière entre -32768 et 32767.	Construit à l'aide d'une série de chiffres.
Chaîne de caractères	*'texte'*	Une chaîne de caractères délimitée par des apostrophes.	N'importe quoi peut être mis entre les deux apostrophes sauf une autre apostrophe. Pour en insérer une il faut la précéder d'un '\'. D'autres caractères spéciaux peuvent être ajoutés ainsi : \n → fin de ligne \t → tabulation \\ →un seul \ \' →un appostrophe
commentaire	// /* */	Ignorer les caractères jusqu'à la fin de la ligne. Ignorer les caractères entre les /* et */ inclus.	N'importe quoi peut être mis dans un commentaire. Les commentaires ne sont pas imbricables.
Séparateur	, ;	Séparateur d'éléments	
opérateur	+ - / *	Opérateur numérique	voir Expressions
comparateur	< > = <= >= <>	Comparateur numérique	voir Expressions
affecteur	:=	Affectation	voir Instructions
dimension	!	Dimension de collection	voir objet **Stack**
indice	[ ]	Élément de collection	voir objet **Stack**
groupe	| |	Délimiteur de groupe	voir syntaxe des constantes GROUP

Tableau 2 - Unités lexicales

2.1.2 Le pré-processeur

Le pré-processeur du CDL est basé sur celui de C. Les directives disponibles sont décrites tableau 3. Il y a certaines différences avec celui de C :

- Les noms de fichiers sont entre apostrophes comme les chaînes de caractères en CDL.
- Tout symbole, variable, nom de procédure, etc. peut être utilisé dans les clauses conditionnelles. Cette particularité vient du fait que, contrairement au C, le pré-processeur et le langage partagent la même table de symboles lors de la compilation, car ils sont chaînés à la suite de l'analyseur lexical, plutôt qu'utilisés en passes successives.
- Le pré-processeur CDL est limité à 16 niveaux de récursion.

#include 'file'	insère le texte constituant le fichier 'file' au point courant dans le fichier source de la compilation. Le fichier 'file' est pris dans le répertoire d'inclusion.
#define n	donne au symbole n l'état défini, au niveau du pré-processeur seulement. L'état d'un symbole peut être : défini ou non-défini.
#ifdef n	compile les lignes de textes suivantes jusqu'au prochain **#else** ou **#endif** si n a été auparavant défini (par un **#define** ou dans le code).
#ifndef n	compile les lignes de textes suivantes jusqu'au prochain **#else** ou **#endif** si n n'a pas été auparavant défini (par un **#define** ou dans le code).
#else	change l'état de la compilation (compiler ou ne pas compiler) si utilisé entre un **#ifdef** ou un **#ifndef** et un **#endif**.
#endif	enlève la restriction sur la compilation engendrée par le **#ifdef** ou **#ifndef** précédent.

Tableau 3 - Directives du pré-processeur CDL

Le choix de pré-processeur vient du fait que :

1. Le PASCAL de base, dont la syntaxe de CDL est dérivée, n'offre pas de directives de pré-compilation. Il devient impossible d'utiliser (et surtout de réutiliser) divers fichiers pour bâtir un projet complet.

2. Le pré-processeur C est bien connu.

2.2 Expressions

CDL est un langage typé où chaque expression composée hérite son type de celui de ses éléments et de l'application d'un opérateur. Ces éléments peuvent être des constantes nommées ou littérales ou des variables. Ces éléments peuvent être globaux (accessibles à tous), locaux (accessibles à la procédure, fonction ou prédicat en définition) ou contextuels à une instance (accessibles aux méthodes de l'instance). Cette section décrit la syntaxe et la sémantique des expressions en CDL. Elle est divisée en sous-sections décrivant les composants d'une expression. La figure 5 décrit les expressions CDL en format BNF.

```
expression ::= expression ('+' | '-' | '*' | '/' | AND | OR | MOD | '<<' | '>>') expression |
               expression ('=' | '<>' | '<=' | '>=') expression |
               ('+' | '-' | NOT) expression |
               id |
               integer |
               '(' expression ')' |
               [ id ] ! |
               [ id ] '[' expression [ '..' expression ] ']'
```

Figure 5 - BNF d'une expression

2.2.1 Types

CDL contient certains types de base classiques, comme INTEGER, BOOLEAN et STRING, certains types fondés sur un type de base comme INDEX ou CARD et deux types de "pointeurs", GROUP et pointeur d'instance. La figure 6 décrit les types CDL sous la forme BNF. La sémantique de chaque type est décrite au tableau 4, où l'on voit aussi quels types sont compatibles, c'est-à-dire où une variable d'un type peut recevoir une valeur de l'autre type et vice-versa.

```
types ::= INDEX | CARD | INTEGER | BOOLEAN | STRING | GROUP | object_name
object_name ::= id
```

Figure 6 - BNF des types

Type	Exemple	Compatibilité	Sémantique
INDEX	2, 5, etc.	INTEGER	Un entier pouvant servir d'index à une collection (**Stack**)
CARD	1, 2, etc.	INTEGER	Un entier pouvant servir d'élément à une collection (**Stack**)
INTEGER	3, 5, etc.	INDEX, CARD	Une valeur entière
BOOLEAN	TRUE, FALSE		Une valeur d'état
STRING	'chain', 'a', etc.		Une chaîne de caractères
GROUP	|inst1, inst2|, ||, etc.		Une liste d'instances
object_name	Stack, Cards, etc.		Un nom d'objet permettant de déclarer un pointeur sur une instance de ce type

Tableau 4 - Sémantique des Types

Une particularité intéressante est que chaque nouvel objet défini devient un représentant qui peut servir à déclarer des variables analogues[9]. Celles-ci pointent vers une instance de l'objet, c'est-à-dire font référence à une instance sans en créer de nouvelles. Par exemple, à la figure 7 est défini un objet fenêtre (**Window**) puis une procédure **`ShowDraw`** qui a en paramètre une instance de celui-ci et qui appelle les méthodes **`Show`** et **`Draw`** de cette instance. La déclaration du paramètre **`v`** de type **Window** ne crée pas d'instance, elle fait plutôt référence à une instance créée précédemment dans une autre partie du programme.

[9] **Stack**, **Cards** et **Window** en sont un exemple. Ils sont décrits implicitement dans l'expression BNF des types, à travers la règle ***object_name***

```
object Window is
  const parent : Window,
        X : integer := 0,
        Y : integer := 0,
       ...
end Window;

procedure ShowDraw( v : Window ) is
     begin v.Show; v.Draw; end;
```

Figure 7 - Exemple de type objet

Pour continuer l'exemple, à la figure 8 est définie une instance **MainWindow** de l'objet **Window** et la procédure **Start** appelle **ShowDraw** en donnant comme instance de référence **MainWindow**. La procédure **ShowDraw** appliquera donc les méthodes **Show** et **Draw** à **MainWindow**;

```
Window MainWindow is
     ...
end MainWindow;

procedure Start is
     begin
     ShowDraw( MainWindow );
     ...
     end;
```

Figure 8 - Exemple de liaison de pointeur d'instances

2.2.2 Constantes

Il existe deux modes d'utilisation des constantes dans une expression. On peut utiliser des constantes nommées, référençables par un identificateur unique, ou littérales transformées en valeurs réelles par l'analyseur lexical.

Les constantes littérales sont décrites tableau 2 dans les entrées : entier, chaîne de caractères et groupe. Ce sont les seules et elles sont typées par l'analyseur lexical.

La figure 9 définit les constantes nommées en BNF. ***expression_without_var*** est une expression où chaque ***id*** présent correspond à une constante, littérale ou nommée, et non à une variable, ce qui rend l'évaluation statique possible. Le type de l'expression détermine celui de la constante. C'est pourquoi, contrairement à la définition de variables, aucun type explicite n'est spécifié.

const_def ::= CONST ***const_elm*** (',' ***const_elm***)* ';'
const_elm ::= ***id*** ':=' ***const_exp***
const_exp ::= ***expression_without_var***

Figure 9 - BNF de la définition de constantes

Des constantes nommées peuvent aussi être définies dans un objet. Leurs valeurs sont locales aux méthodes par défaut de l'objet et aux redéfinitions de méthodes dans les

instances. Ces constantes peuvent aussi être redéfinies au niveau de chaque instance au moment de la définition.

2.2.3 Variables

Les variables en CDL peuvent être créées de quatre façons. Dans chaque cas la déclaration est formée de deux éléments, un identificateur et un type. L'identificateur devient la référence, de portée différente selon le type de création, alors que le type sera utilisé lors de la création ou lors de l'analyse d'expression pour déterminer le type résultant. Les quatre façons sont :

1. variable globale : une variable définie au début du programme et connue des constructions subséquentes.
2. variable d'instance : une variable définie dans un objet, ayant un espace mémoire distinct dans chaque instance et connue des méthodes de l'objet ou de l'instance.
3. variable locale : une variable définie au début d'une construction (procédure, fonction ou prédicat). N'est connue que dans celle-ci.
4. paramètres : une variable définie au début d'une construction et ayant une valeur fournie par l'appelant. N'est connue que dans celle-ci.

La figure 10 donne un exemple de chacune. À la première ligne est définie une variable globale `c`. Puis, lors de la définition de l'objet **Stack**, une variable **`modified`** est définie. Chaque instance de l'objet **Stack** possédera une copie de cette variable et pourra

la modifier à son gré. Finalement la fonction **countBits** décrit les deux dernières façons, la variable **b** est définie comme paramètre, elle recevra une valeur fournie par l'appelant au moment de l'exécution. La variable **a** est définie comme variable locale à la fonction **countBits**. Les variables **a** et **b** seront créées lors d'un appel de fonction à **countBits** et seront détruites à la sortie de la fonction.

```
var c : integer;

object Stack is
     ...
     var modified : boolean;
     ...
end Stack;

function countBits( b : integer ) : integer is
     var a : integer;
     begin
     ...
     end;
```

Figure 10 - Exemple de déclaration de variables

La figure 11 décrit en BNF la syntaxe de définition des variables globales, locales ou d'instances. Pour la définition de paramètres, la syntaxe est identique hormis l'absence du mot clef VAR.

var_def ::= VAR (*var_elm* ',')*
var_elm ::= *id* (',' *id*)* ':' *types*

Figure 11 - BNF de la définition de variables

Une variable prédéfinie **self** est implicitement présente dans chaque méthode d'un objet. Elle est du type de l'objet et elle pointe sur l'instance courante au moment de l'évaluation[10]. Par exemple, à la figure 12 la deuxième définition explique ce qui se passe implicitement lors de la première.

```
Window procedure Show is
     begin ... end;

Window procedure Show is
     var self : Window;
     begin ... end;
```

Figure 12 - Variable self implicite

2.2.4 Opérateurs

CDL utilise les opérateurs numériques et logiques traditionnels, addition, soustraction, comparaison numérique, etc. Trois nouveaux sont introduits : pour obtenir le nombre d'éléments d'une collection (**Stack**), un élément de cette collection, une série d'éléments de cette collection. Le tableau 5 décrit les opérateurs. Notons que les opérateurs AND et OR s'appliquent différemment en fonction du type de sous-expressions présentes, entières ou booléennes. Notons aussi que la barre verticale (|), délimiteur de groupe, n'est pas un opérateur mais un constructeur statique pour les constantes de type GROUP. Le tableau 6 décrit la priorité des opérateurs.

[10] les termes évaluation et éxécution sont ici synonymes

Opérateur(s)	Exemple	Sémantique
+ - * /	12 + 6 → 18 12 - 6 →6 12 * 6 → 48 12 / 6 → 2	Addition, a + b Soustraction, a - b Multiplication, a * b Division, a / b
MOD AND OR	13 MOD 6 →1 7 AND 3 → 3 4 OR 3 → 7	Modulo, a MOD b, le reste de la division de a par b Et numérique Ou numérique
NOT AND OR	NOT TRUE → FALSE TRUE AND TRUE → TRUE TRUE OR FALSE → TRUE	Négation logique, $\neg$ a Et logique, a $\cap$ b Ou logique, a $\cup$ b
>> <<	15 >> 3 →1 3 << 3 →24	Décalage binaire vers la droite, ou a >> b est équivalent à $\lvert a / 2^b \rvert$ Décalage binaire vers la gauche, ou a << b est équivalent à $a * 2^b$
= <> < > <= >=	12 = 11 → FALSE 12 <> 11 → TRUE 12 < 11 → FALSE 12 > 11 → TRUE 12 <= 11 → FALSE 12 >= 11 → FALSE	Equivalence numérique, a = b Non-équivalence numérique, a ≠ b Comparaison numérique, a plus-petit-que b Comparaison numérique, a plus-grand-que b Comparaison numérique, a plus-petit-ou-égal-à b Comparaison numérique, a plus-grand-ou-égal-à b
!	instance_stack_a! → 12	**a**!, nombre d'éléments d'une instance **a** d'un objet collection (**Stack**)
[ ]	instance_stack_a[3] → 23	**a**[**n**], élément d'indice **n** d'une instance **a** d'un objet collection (**Stack**)
[..]	instance_stack_a[3..6] → 12,23,34,13	**a**[$\mathbf{n}^1$ .. $\mathbf{n}^2$], éléments d'indices $\mathbf{n}^1$ à $\mathbf{n}^2$ (inclus) d'une instance **a** d'un objet de collection (**Stack**)

Tableau 5 - Sémantique des opérateurs

haute priorité	
NOT '+' '-'	*opérateurs unaires préfixés*
'!' '[..]'	*opérateurs unaires postfixés*
AND MOD '*' '/' '>>' '<<'	
OR '+' '-'	
'=' '<>' '<=' '>='	
basse priorité	

Tableau 6 - Priorité des opérateurs

2.3 Objets

Un objet définit un moule, ou une catégorie de prototype, à partir duquel nous pouvons créer des individus. Au minimum celui-ci doit comporter un nom et une méthode de création. En CDL, la méthode de création (**`Register`**) est ajoutée à l'objet de façon implicite lors de sa définition. Le nom de l'objet devient la référence pour la création d'instances, ou prototypes, correspondants. Il est aussi utilisé pour créer des variables de pointeurs d'instances. La figure 13 décrit la syntaxe BNF de la définition d'objets.

object_def ::=	OBJECT ***id*** IS ***object_body*** END ***id*** ';'
object_body ::=	(***const_init*** | ***var_init*** | ***meth_init*** | ***multi_init***)*
const_init ::=	CONST ***id*** ':' ***types*** [':=' ***const_exp***] ';'
var_init ::=	VAR ***id*** ':' ***types*** [':=' ***const_exp***] ';'
meth_init ::=	(PREDICATE | PROCEDURE | FUNCTION) ***id*** [***parms***] [IS ***statement***] ';'
multi_Init ::=	***id*** IS ***id*** (',' ***id***)* ';'

Figure 13 - BNF de la définition d'objet

Quatre types de constructions peuvent apparaître dans un objet : constante, variable, méthode et interface multiple de méthodes. La syntaxe de définition est sensiblement la même que leurs homonymes globaux et locaux mais ils sont empaquetés dans l'objet. Les méthodes d'objet, non re-définissables par instance, sont décrites section 2.5.

Les constantes et les variables sont typées et peuvent avoir une valeur initiale par défaut. Ces valeurs peuvent être modifiées lors de la définition d'instances. La seule différence entre constante et variable se situe sémantiquement au niveau du code et non de la définition. Un identificateur classé comme constante ne peut se voir affecter une valeur. Une constante qui a une valeur par défaut dans l'objet peut s'en voir affecter une autre dans l'instance au moment de la définition. Par exemple, à la figure 14, les constantes **`parent`**, **`X`**, **`Y`**, **`W`** et **`H`** sont définies et initialisées, la constante **`text`** est définie mais non-initialisée et une variable **`modified`** est définie et initialisée à **`TRUE`**.

Les méthodes sont des procédures, fonctions ou prédicats définis à l'intérieur de l'objet. Elles peuvent avoir un comportement par défaut, utilisé par les instances de l'objet lorqu'elles ne sont pas redéfinies, ou être vides, n'attendant qu'à être spécialisées. Les procédures **`Init`**, **`Select`**, **`UpdateFrench`** et **`UpdateEnglish`** de la figure 15 sont dans cette situation. La procédure **`Register`** à déjà un comportement par défaut[11]. Elles possèdent toutes une variable **`self`** implicite discutée précédemment.

[11] qui appelle un module externe de Windows pour la création physique du bouton

Les interfaces multiples de méthodes permettent de voir le même objet de plusieurs façons. Elles sont composées d'un identificateur, l'interface, et d'une série de méthodes à définir. Lorsqu'une instance est déclarée. si un comportement est affecté à l'interface, il sera copié dans chaque méthode définie par l'interface. Les interfaces multiples n'ont donc pas d'entrées dans la table virtuelle de l'instance. Elles vont plutôt affecter les entrées des méthodes qu'elles référencent. Une méthode multiple peut être appelée ce qui se transformera en un appel à la première entrée de sa liste.

En résumé, certaines parties du même objet peuvent être vues simultanément de façon différente à l'extérieur de l'objet mais une seule des interfaces a besoin d'être codée par le programmeur. On peut voir cela comme du sucre syntaxique servant à économiser du temps au programmeur en limitant le nombre de fois qu'il devra recopier le même code lorsque plusieurs méthodes d'un même objet effectuent la même chose.

Par exemple, nous avons ajouté figure 14 du multilinguisme à l'objet bouton avec des méthodes **`UpdateEnglish`** et **`UpdateFrench`** et une méthode multiple **`Update`**. L'objet peut donc ensuite être vu de deux façons : unilingue si on utilise **`Update`** ou bien bilingue si on utilise **`UpdateEnglish`** et **`UpdateFrench`**.

```
object Button is
  const parent : Window := MainWindow,
        X : integer := 0, Y : integer := 0,
        W : integer := 0, H : integer := 0,
        text : string;
  var modified : boolean := FALSE;
  procedure Register is begin asmx self; asm MSG 5 0; end;
  procedure Init;
  procedure Select;
  procedure UpdateEnglish;
  procedure UpdateFrench;
  Update is UpdateFrench, UpdateEnglish;
end Button;
```

Figure 14 - Objet bouton

Si **Update** est spécialisé dans une instance, **UpdateEnglish** et **UpdateFrench** auront le même comportement parce que les entrées de la table virtuelle pour ces dernières pointeront vers la méthode de spécialisation. L'instance figure 15 montre ceci. Si un appel à **Update** du bouton **Deal** est effectué quelque part dans le code, la méthode **UpdateFrench** de celui-ci sera appelée.

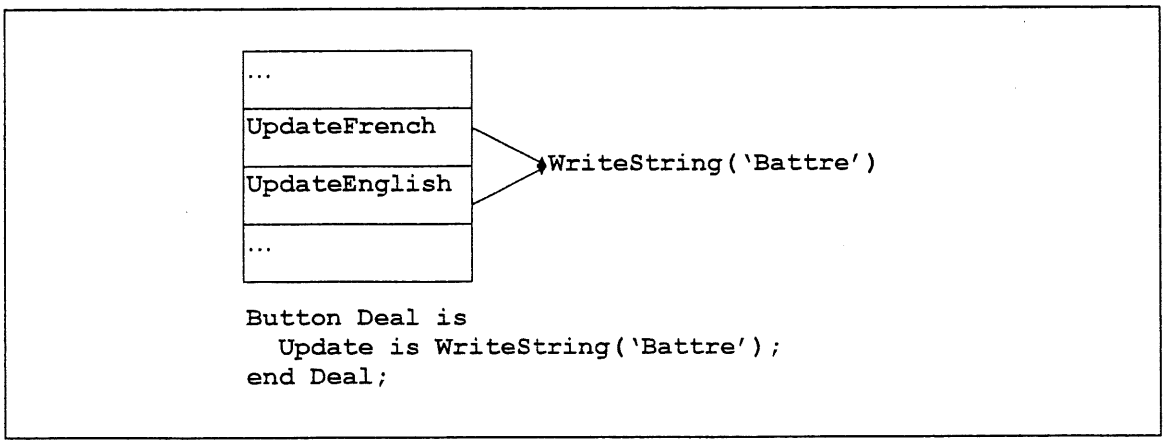


Figure 15 - Méthode multiple multilingue

2.4 Instances

Les instances sont en même temps des individus animés et des prototypes pouvant en créer d'autres. Une instance peut provenir directement du moule primaire (l'objet), d'un moule inanimé spécialisé (l'instance virtuelle) ou d'une de ses soeurs. Elle peut en plus, à la définition, changer un de ses comportements ou la valeur d'une de ses constantes. Pour changer un comportement, elle peut en définir un nouveau textuellement ou aller en chercher un dans une autre instance, possiblement virtuelle. La figure 16 est la syntaxe BNF de la définition d'instances.

Beaucoup de liberté et de possibilités ont été développées au niveau de la définition d'instances pour permettre d'écrire du code rapidement et de garder celui-ci concis. Le prix à payer est que dans le cas de gros systèmes, le changement de comportement d'une instance peut avoir des répercussions sur plusieurs autres instances et ce à travers des chemins très tortueux. Un effort devra être fait à ce niveau par les créateurs de ce système pour réduire l'utilisation des moyens de définition d'instances à un sous-ensemble plus sécuritaire.

instance_def ::=	***object_name id*** [FROM ***id2***] IS ***stack_body*** END ***id*** ';'
object_name ::=	***id***
stack_body ::=	(***const_init*** | ***meth_init***)*
const_init ::=	***id*** ':=' ***const_exp*** ';'
meth_init ::=	***id*** [***parms***] (FROM ***id2*** | IS ***statement***) ';'

Figure 16 - BNF de la définition d'instances

L'exemple, figure 17, illustre chacune des possibilités de définition pour les instances. Le bouton **Help** est créé à partir du bouton **Deal**. Les constantes **X**, **Y**, **W** et **H** sont définies pour placer le bouton dans le coin supérieur gauche de la surface. La méthode **Init** est définie pour rendre le bouton visible et **Select** est copiée de l'instance **HelpModuleBtn**. Les méthodes **UpdateFrench** et **UpdateEnglish** n'étant pas redéfinies, elles proviendront de **Deal**. **Register** garde sa valeur par defaut spécifié lors de la définition de l'objet bouton.

```
Button Help from Deal is
  X := 1; Y := 1;
  W := 5; H := 2;
  Init is Show;
  Select from HelpModuleBtn;
end Help;
```

Figure 17 - Instance de bouton

Les instances virtuelles, inanimées, sont créées de la même façon avec en plus le mot clef VIRTUAL devant le nom de l'objet. La figure 18 décrit ceci en BNF.

virtual_instance_def ::= VIRTUAL *instance_def*

Figure 18 - BNF de la définition d'instances virtuelles

2.4.1 Créations d'instances

Une instruction permet d'instancier : ORDER. De plus, elle force un ordre de création qui sera nécessaire, comme vu avec Canfield. L'instanciation causera l'appel de la méthode implicite **`Register`** de l'instance. Une décision d'émettre implicitement une instruction ORDER pour les instances définies mais oubliées à la fin de la compilation nous a forcé à ajouter le concept d'instance virtuelle. Les instances non instanciées par ORDER auraient dû être par défaut des instances virtuelles. La figure 19 est la syntaxe de l'instruction. Les instances virtuelles ne peuvent évidemment pas être spécifiées.

order_def ::= ORDER *instance* (',' *instance*)*
instance ::= *id*

Figure 19 - BNF de la création d'instances

2.5 Fonctions et Instructions

Les fonctions CDL ressemblent à celles de PASCAL. Il y a des procédures ne retournant pas de valeur et des fonctions retournant une valeur de type prédéterminé. Une forme spéciale de fonction créée pour CDL, le prédicat, retourne une valeur booléenne. C'est celle utilisée pour définir les post-conditions et les conditions de fin de programme.

```
proc_def ::=  [ object_name ] PROCEDURE id [ parms ] IS [ var_def ] statement ';'
func_def ::=  [ object_name ] FUNCTION id [ parms ] ':' types
              IS [ var_def ] statement ';'
pred_def ::=  [ object_name ] PREDICATE id [ parms ] IS [ var_def ] statement ';'
parms ::=     '(' var_list (';' var_list)* ')'
```

Figure 20 - BNF de la définitions de fonctions

La syntaxe est décrite figure 20. ***parms*** définit les paramètres, variables locales à la construction initialisées par l'appelant, ***var_def*** définit des variables locales à la construction, initialisées à la valeur non-définie. ***Object_name*** dénote une méthode d'objet, c'est-à-dire une méthode globale unique, toujours exécutée dans le contexte d'une instance. Le ***statement*** qui est le corps de la fonction peut être simple, c'est-à-dire non délimité par un BEGIN - END.

```
statement ::=  id ':=' expression | id '.' id ':=' expression |
               id [ parms ] | id '.' id [ parms ] |
               BEGIN (statement)* END |
               IF expression THEN statement [ ELSE statement ] |
               WHILE expression DO statement
               BREAK [ PROCEDURE ] |
               RETURN expression |
               WAIT expression [ id ] |
               ON id DO statement |
               WITH id : type DO statement FOR expression (',' expression)* |
```

Figure 21 - BNF des instructions

La syntaxe des instructions est décrite figure 21. Seules celles créées ou nuancées pour CDL sont décrites en détail dans les sous-sections suivantes.

Affectation et Appel de fonctions

Les affectations et les appels de fonction ont la même forme qu'en PASCAL mais ils peuvent être préfixés d'un identificateur d'instance. L'opération est alors effectuée dans le contexte de celle-ci.

L'ordre de recherche pour la résolution des identificateurs de variables non préfixés est le suivant : variables locales, paramètres, variables d'instance puis variables globales. Pour les identificateurs de fonctions : méthodes d'instance, méthodes d'objet puis fonctions globales. L'objet contextuel n'est préservé que dans les deux premiers cas. Si les identificateurs sont préfixés, la recherche est limitée à l'instance et à l'objet.

BREAK et RETURN : terminateur de bloc

Ces instructions forcent la fin de l'exécution d'un bloc. RETURN sert à terminer une fonction ou un prédicat et à retourner la valeur résultante à l'appelant. BREAK, seul, sert à terminer l'exécution du ON ou WITH courant. Dans le cas du WITH, l'itération courante se termine immédiatement et les éléments non traités sont oubliés. BREAK postfixé du mot PROCEDURE est utilisé à l'intérieur d'une procédure pour en sortir. C'est l'équivalent du RETURN des fonctions.

WAIT : Retardateur

Cette instruction peut être utilisée de deux façons mais toujours en forçant un délai. Si un nom de procédure est spécifié, la transaction courante est terminée, le délai effectué, puis la procédure est appelée créant ainsi une transaction. Si seul un délai est spécifié, l'exécution est immédiatement suspendue.

ON - DO : changement de contexte

Cette instruction permet d'exécuter un ***statement*** dans le contexte d'une instance. Les variables de la construction courante sont disponibles dans le corps du ON - DO ainsi que celles de l'instance spécifiée. Cette instruction sert généralement à effectuer une série d'affectations ou d'appels de fonction sans avoir à les préfixer d'un identificateur d'instance.

WITH - DO : Itérateur

Comme vu, cette instruction sert à effectuer une opération sur plusieurs instances d'objets ou de types de base. La variable définie dans l'en-tête n'est visible que dans le corps de l'instruction et est associée, à chaque itération, à l'instance spécifiée. Si un élément de type GROUP est donné en paramètre, chaque élément du groupe sera utilisé récursivement pour une itération. Le code contenu dans un WITH reste dans le contexte courant, contrairement à celui du ON - DO.

Le code figure 22 est valide, ce qui fait presque de "6" une instance de l'objet INTEGER.

```
WITH A : integer DO
     B := B + a
FOR 6, 5, 2, 7;
```

Figure 22 - Itération sur des entiers

Chapitre 3

Implantation de CWS

Ce chapitre décrit l'implantation du compilateur et de l'interprète. Dans la première section nous verrons le compilateur, la table des symboles, le format des données et la production de code. Ensuite nous traiterons de l'interprète, ses registres et ses instructions. Nous finirons par une description des objets prédéfinis, fenêtre, cartes, pile, bouton et port.

3.1 Compilateur CWS

Le compilateur CWS est fondé sur les principes de compilation de Wirth dans [Wir75] et sur le typage décrit par Aho [Aho89]. Comme déjà spécifié, il comporte un pré-processeur analogue à C. L'analyse syntaxique est LL(1), c'est-à-dire sans essai-erreur et descendante.

Le pré-processeur s'insère entre l'analyseur lexical, **getsym**, et l'analyseur syntaxique. Si une construction du pré-processeur est reconnue par **getsym**, le pré-processeur prend contrôle de l'analyse jusqu'à la fin de la construction. Contrairement à C, aucune passe de pré-traitement n'est ajoutée.

Nous n'avons pas implanté de reprise sur erreur car dans l'environnement CWS, quand une erreur de compilation survient, le fichier fautif est automatiquement ouvert en édition et la construction fautive est mise en évidence. La compilation étant rapide et les programmes CDL assez petits, ceci n'a pas créé de problème jusqu'à maintenant. La compagnie Borland a fait le même choix pour Turbo Pascal.

3.1.1 Table des symboles

La table des symboles permet de mémoriser et de vérifier les attributs sémantiques, genre et type, des identificateurs. Les différents genres sont décrits tableau 7. Ceux pour un objet, procédure, fonction et prédicat, forment ce qu'on appelle les méthodes d'objet, c'est-à-dire des fonctions spécifiques à chaque objet, non redéfinissables par instance. Comparativement, les genres pour une instance, procédure, fonction et prédicat forment ce qu'on appelle les méthodes d'instance, c'est-à-dire des fonctions redéfinissables pour chaque instance. Les entrées qualifiées de genre d'instance possèdent un espace dans la table virtuelle de l'instance; elles sont ajoutées à la table des symboles lors d'une définition d'objet. L'exemple de code, figure 23, montre les identificateurs du tableau 7 en contexte, sauf pour les genres procédures, fonctions et prédicats dont un seul est illustré.

Genre	Exemple	Description
TYPE	`integer`	un des types prédéfinis de CDL
OBJET	`Stack`	un objet
INSTANCE	`A1`	une instance
INSTANCE VIRTUELLE	`standard_f`	une instance virtuelle
CONSTANTE	`turnover`	constante globale
VARIABLE	`players`	variable globale
PROCÉDURE		procédure globale, aucun contexte
FONCTION	`CountBits`	fonction globale, aucun contexte
PRÉDICAT		prédicat global, aucun contexte
PROCÉDURE D'OBJET		procédure globale, contexte d'un objet
FONCTION D'OBJET	`isEmpty?`	fonction globale, contexte d'un objet
PRÉDICAT D'OBJET		prédicat global, contexte d'un objet
CONSTANTE D'INSTANCE	`X`	constante d'objet redéfinissable par instance
VARIABLE D'INSTANCE	`modified`	variable d'objet redéfinissable par instance
PROCÉDURE D'INSTANCE	`Start`	procédure d'objet redéfinissable par instance
FONCTION D'INSTANCE		fonction d'objet redéfinissable par instance
PRÉDICAT D'INSTANCE		prédicat d'objet redéfinissable par instance
MULTI-INIT	`SelectTo`	interface multiple

Tableau 7 - Genres dans la table des symboles

```
object Stack is
  const X : integer;
  var modified : boolean;
  procedure Start is …;
  SelectTo is SelectLeftTo, SelectRightTo;
end Stack;

Stack predicate isEmpty? Is
  RETURN (! = 0);

function CountBits( a : integer ): integer is
  …;

const turnover := 3;
var players : integer;

virtual Stack standard_f is
…
end standard_f;

Stack A1 from standard_f is
…
end A1;
```

Figure 23 - Exemple de genres

Lors de l'analyse, on ajoute les symboles du bloc courant à la table et on les enlève lorsqu'on sort du bloc. Ce qui veut dire que la table est une pile où seuls sont visibles les symboles référençables. Chaque entrée comporte les éléments suivants :

- un identificateur
- un genre
- un type, pour les variables, les constantes et les fonctions
- une valeur pour les constantes

- le niveau d'imbrication de l'entrée, 0 pour les constructions globales, 1 pour les locales (internes)
- un indice, absolu ou relatif, pour accéder à des valeurs sur la pile ou une adresse absolue dans la mémoire de code
- un pointeur vers une autre entrée de la table qui contient l'objet père, au besoin.

Voici un exemple de la table lors de l'analyse du code suivant :

```
object Stack is
  const X : integer;
        Y : integer := 0;
  procedure Start is
    var a, b : integer;
1 : Stack, objet, niveau 0
2 : X, constante d'instance, adresse relative dans l'instance : 0, père est 1 (Stack)
3 : Y, constante d'instance, adresse relative dans l'instance : 1, valeur : 0,
      père est 1 (Stack)
4 : Start, procédure d'instance, adresse dans l'instance : 2,
      père est 1 (Stack)
5 : a, variable, type integer, niveau 1, adresse relative dans la pile : 4
6 : b, variable, type integer, niveau 1, adresse relative dans la pile : 5
    begin
    ...
    end;
end Stack;

1 : Stack, objet, niveau 0
2 : X, constante d'instance, adresse relative dans l'instance : 0, père est 1 (Stack)
3 : Y, constante d'instance, adresse relative dans l'instance : 1, valeur : 0,
      père est 1 (Stack)
4 : Start, procédure d'instance, adresse dans l'instance : 2, père est 1 (Stack)
```

```
Stack predicate isEmpty? is
  RETURN (self! = 0);

Stack A1, A2;
```

1 : Stack, objet, niveau 0
2 : X, constante d'instance, adresse relative dans l'instance : 0, père est 1 (Stack)
3 : Y, constante d'instance, adresse relative dans l'instance : 1, valeur : 0, père est 1 (Stack)
4 : Start, procédure d'instance, adresse dans l'instance : 2, père est 1 (Stack)
5 : isEmpty?, prédicat d'objet, adresse du code : 57, père est 1 (Stack)
6 : A1, instance, objet Stack, adresse absolue dans la pile : 34
7 : A2, instance, objet Stack, adresse absolue dans la pile : 44

3.1.2 Format des valeurs sur la pile

Les données de chaque instance sont contenues dans la première moitié de la pile. Cette partie, de grandeur fixe, est réservée au départ du programme et n'est dépilée qu'à la sortie. Elle est produite à la compilation et contient aussi les variables globales au programme. La deuxième partie, de grandeur variable, contient les contextes de procédures[12]. Il s'agit de l'information nécessaire à garder lors d'un appel de fonction pour pouvoir revenir à l'état précédent à la fin de l'appel.

Les instances utilisent une série de cellules consécutives appelées table virtuelle de l'instance, contenant les constantes, variables et pointeurs de méthodes. Le pointeur d'objet contextuel du contexte de procédure pointe sur la première cellule de cette table.

[12] cadre ou bloc d'activation

Les méthodes d'objet n'occupent aucune place dans l'instance. Celles-ci étant globales aux instances, un seul pointeur est gardé et utilisé pour la génération d'appels. Lorsqu'une instance est créée à partir d'une autre, les cellules de cette dernière sont copiées dans l'espace reservé pour celle-ci.

Un contexte de procédure est une série de cellules consécutives allouées dynamiquement sur la pile et qui contiennent l'environnement d'une procédure. Chaque procédure appelée crée un contexte de procédure sur la pile. Ce contexte est défini ainsi :

Index	Utilisation
0	Lien statique au père
1	Lien dynamique au contexte de procédure de l'appelant
2	Adresse de retour
3	Adresse de l'objet contextuel

Sur la pile, il sera précédé de la valeur de retour et des paramètres effectifs accessibles avec un indice négatif dans le contexte courant. Les variables locales seront à sa suite et accessibles avec un indice positif plus grand que trois. Par exemple, la fonction **add** donnera le contexte suivant :

```
fonction add( a : integer, b : integer ) : integer is
  var d, e : integer;
  ...
```

Valeur de retour	-3
Paramètres (a,b)	-1, -2
Contexte de procédure	0, 1, 2, 3
Variables locales (d,e)	4, 5

Lors de la création d'un nouveau contexte de procédure. on calcule le lien statique au père hiérarchique en descendant au besoin dans cette chaîne. Le pointeur du contexte courant est utilisé comme lien dynamique puis l'adresse de retour et une copie de l'objet contextuel courant y sont ajoutées.

3.1.3 Production de code

CDL est assez simple pour que nous puissions séquentiellement produire le code et réserver de l'espace pour une variable à la suite d'un contexte de procédure. Le code est produit sensiblement de la même façon que Wirth pour PL0 [Wir75]. Nous avons choisi deux instructions, *with-for* et *on-do*, et l'appel de méthode pour montrer nos schémas de traduction.

Un problème est la production de code pour *with-for* car nous devons produire des branchements à des adresses inconnues d'avance. Pour simplifier le travail, nous gardons en mémoire le code produit et nous complétons les branchements à ces adresses lorsque nous les connaissons. Nous devons pouvoir ré-exécuter le corps de l'itération une fois pour chaque élément. Voici le plan de transformation pour *with-for* :

with V *do* S *for* Vs :		saut inconditionnel à L1, JMP L1
	L2 :	code pour effectuer S, avec V comme paramètre
		retour, OPR 0,0
	L1 :	code pour évaluer Vs_1
		appel inconditionnel à L2, CAL L2
		code pour évaluer Vs_2
		appel inconditionnel à L2, CAL L2
	...	

Le corps du *with-for* est considéré comme une procédure d'un paramètre **V**. Un appel de procédure est effectué pour chaque identificateur **V** dans la liste **Vs**. Si un des paramètres de la liste **Vs** est de type GROUP, un appel de fonction est effectué pour chacun de ses éléments.

Pour *on-do* nous devons pouvoir exécuter une instruction dans le contexte d'une instance. Pour ce faire, nous empilons le pointeur de celle-ci, l'échangeons avec l'objet contextuel courant, exécutons l'instruction et replaçons l'ancien contexte. De façon plus formelle :

on V *do* S :	empile pointeur d'instance V
	échange avec contexte courant, EXG 0,3
	code pour effectuer S
	remettre contexte original, STO 0,3

L'instruction *on-do* est la forme explicite du passage de message : instance point message. Le même mécanisme de production de code est utilisé pour les deux. Par exemple, les deux bouts de codes suivants sont sémantiquement équivalents :

```
on instance1 do
  begin
  PULL 5 FROM self TO instance2;
  self.n = self.n + 5;
  begin
```

```
Stack instance1 is
  aMethod is
    begin
    PULL 5 FROM self TO instance2;
    self.n = self.n + 5;
    begin
end instance1;

...

  instance1.aMethod
```

Dans le premier cas, le code à éxécuter est écrit directement dans le corps du *on-do*, tandis que dans l'autre cas une méthode (**aMethod**) sémantiquement semblable est écrite et un appel sous forme de passage de message est effectué ailleurs.

3.2 Interprète

Notre machine virtuelle, inspirée de celle de Wirth [Wir75], comporte trois espaces de mémoire distincts, un espace instructions, une pile et un espace de chaînes de caractères. L'espace instructions contient le code et n'est jamais changé après le "chargement"[13]. La

[13] c'est un espace "read-only"

pile contient les variables et les instances, chargées à partir du fichier exécutable, et les valeurs intermédiaires. Les opérateurs arithmétiques opèrent sur le ou les deux éléments au sommet de la pile et les remplacent par le résultat[14]. L'espace de chaînes de caractères contient les chaînes statiques définies lors de la compilation et celles créées lors de l'exécution. Les éléments dynamiques sont alloués sur la pile, aucun "heap" externe n'existe.

Notre machine possède trois registres :

- T : registre du haut de pile. Pointe sur l'élément au sommet de la pile.
- I : registre des instructions. Contient l'instruction en cours d'interprétation.
- B : registre de contexte de procédure. C'est un indice dans la pile.

Comme les variables sont allouées dynamiquement sur la pile, on ne peut déterminer l'adresse absolue de celles-ci. Nous utilisons un mode d'adressage relatif. Une variable sera trouvée en additionnant un déplacement au pointeur de contexte de la procédure contenant cette variable. Si la variable est dans le contexte courant, alors le registre B nous donne ce pointeur. Sinon nous descendons dans la chaîne des liens statiques.

Une instruction comporte trois champs :

1. Un mnémonique de trois caractères, comme LIT, OPR, LOD, etc.

[14] l'ordre d'évaluation est donc post-fixé

2. Un paramètre **l** qui représente le nombre de niveaux de recul dans la chaîne des contextes de procédures ou, dans quelques cas, une modification du sens de l'instruction.

3. Un paramètre **a** qui représente une valeur ou une adresse absolue ou relative.

Les sous-sections suivantes décrivent les quatre types d'instructions de la machine virtuelle : opération, chargement, sauvegarde et saut de contrôle. Les instructions ajoutées par rapport à celle de Wirth sont : quelques OPR, LOE, LEA, STE, LEE, EXG, JPC 1, JPP 0, JPP 1 et MSG.

3.2.1 Instructions d'opération

Deux instructions permettent d'effectuer des opérations sur la pile. OPR 0, **a** effectue une opération sur un certain nombre d'éléments au sommet, les enlève puis y ajoute le résultat. Le tableau 8 décrit les opérateurs possibles pour l'instruction OPR. L'instruction INT 0, **a** permet de changer de façon relative la position du pointeur de pile (registre *T*) en y ajoutant **a**. Elle servira notamment à réserver de l'espace sur la pile pour les variables, contextes de procédures et instances, avec un saut positif (**a** > 0) et à éliminer ces espaces avec un saut négatif (**a** < 0).

a	Nom de l'opérateur	Nombre d'arguments	Description
0	Return	0	Dépile le contexte de procédure et met l'adresse de retour dans *I* (pile[*B*+2]→*I*)
1	Neg	1	-pile[*T*] → pile[*T*]
2	Plus	2	T - 1 → T, pile[*T*] + pile[*T*+1] → pile[*T*]
3	Minus	2	T - 1 → T, pile[*T*] - pile[*T*+1] → pile[*T*]
4	Mult	2	T - 1 → T, pile[*T*] * pile[*T*+1] → pile[*T*]
5	Div	2	T - 1 → T, pile[*T*] / pile[*T*+1] → pile[*T*]
6	Modulo	2	T - 1 → T, pile[*T*] mod pile[*T*+1] → pile[*T*]
8	equal	2	T - 1 → T, si pile[*T*] = pile[*T*+1] alors 1 → pile[*T*] sion 0 → pile[*T*]
9	not-equal	2	T - 1 → T, si pile[*T*] = pile[*T*+1] alors 0 → pile[*T*] sion 1 → pile[*T*]
10	less	2	T - 1 → T, si pile[*T*] < pile[*T*+1] alors 1 → pile[*T*] sion 0 → pile[*T*]
11	greater-or-equal	2	T - 1 → T, si pile[*T*] > ou = pile[*T*+1] alors 1 → pile[*T*] sion 0 → pile[*T*]
12	greater	2	T - 1 → T, si pile[*T*] > pile[*T*+1] alors 1 → pile[*T*] sion 0 → pile[*T*]
13	less-or-equal	2	T - 1 → T, si pile[*T*] < ou = pile[*T*+1] alors 1 → pile[*T*] sion 0 → pile[*T*]
14	Not (bool)	1	1 - pile[*T*] → pile[*T*]
15	Not (numérique)	1	¬pile[*T*] → pile[*T*]
16	Et (numérique)	2	T - 1 → T, pile[*T*] ∩ pile[*T*+1] → pile[*T*]
17	Ou (numérique)	2	T - 1 → T, pile[*T*] ∪ pile[*T*+1] → pile[*T*]
18	Décalage droit	2	T - 1 → T, pile[*T*] * $2^{\text{pile}[T+1]}$ → pile[*T*]
19	Décalage gauche	2	T - 1 → T, pile[*T*] / $2^{\text{pile}[T+1]}$ → pile[*T*]
21	Échange	2	Échange les deux valeurs au sommet
22	Inc	1	pile[*T*] + 1 → pile[*T*]
23	Dec	1	pile[*T*] - 1 → pile[*T*]
24	Valeur indirecte	2	Ajoute au sommet la valeur à l'indice calculé en ajoutant les deux paramètres
25	Valeur au hasard	1	Valeur pseudo aléatoire entre 0 et la valeur sur la pile

Tableau 8 - Instruction OPR

3.2.2 Instructions de chargement

Les instructions de chargement ajoutent une valeur sur la pile et n'en enlèvent pas. Trois types existent : chargement de constante, de variable ou d'adresse de variable. Les deux derniers peuvent être appliqués directement sur un espace mémoire calculé en descendant de **l** niveaux dans la chaîne des contextes de procédures et en ajoutant **a** à l'adresse obtenue. Ces instructions sont :

- LIT 0, **a** : empile la valeur **a**.
- LOD **l**, **a** : descend de **l** niveaux dans la chaîne des contextes de procédures et empile la valeur de la cellule à l'indice **a** du contexte de procédure.
- LEA **l**, **a** : descend de **l** niveaux dans la chaîne des contextes de procédures et empile la valeur absolue de l'adresse de la cellule **a** du contexte de procédure.
- LOE 0, **a** : comme LOD mais au lieu de descendre dans la chaîne des contextes de procédures, la valeur de l'objet contextuel du contexte de procédure courant est utilisée (B+3). La valeur à l'indice **a** de l'objet est empilée.
- LEA 0, **a** : comme LEA mais au lieu de descendre dans la chaîne des contextes de procédures, la valeur de la cellule de l'objet contextuel du contexte de procédure courant est utilisée (B−3). L'adresse de la valeur à l'indice **a** de l'objet est empilée.

3.2.3 Instructions de sauvegarde

Trois instructions permettent de prendre une valeur sur la pile et de la sauvegarder ailleurs dans celle-ci. Dans le cas de STO et STE, la valeur au sommet est dépilée. Les instructions sont :

- STO **l,a** : recule de **l** contextes et transfère la valeur au sommet dans la cellule d'indice **a** du contexte de procédure.
- STE 0, **a** : comme STO mais au lieu de reculer dans les contextes, la valeur de l'objet contextuel courant est utilisée (*B*+3). La valeur au sommet est transférée dans la cellule d'indice **a** de l'objet.
- EXG **l,a** : comme STO, sauf que la valeur dans la cellule d'indice **a** du contexte est échangée avec celle du sommet.

3.2.4 Instructions de saut dans le code

Les instructions de saut permettent de changer le flot de contrôle, c'est-à-dire modifier le registre *I*. Quatre types sont possibles : appel avec sauvegarde d'adresse de retour, saut inconditionnel, saut conditionnel et appel à un module externe, écrit dans un autre langage par exemple.

Les trois premières permettant les sauts avec sauvegarde d'adresse de retour sont :

- CAL **l**, **a** : saut à l'adresse **a** (**a** → I). Avant de sauter, un nouveau contexte de procédure est empilé. **l** est utilisé pour savoir quel contexte de la chaîne devient le lien statique du nouveau contexte.
- CAE **l**, **a** : comme CAL, sauf que l'adresse cible est déterminée en prenant la cellule d'indice **a** dans l'objet contextuel (pile[**a** + pile[*B*+3]] → I).
- OPR 0,0 : élimine le contexte du sommet et saute à l'adresse de retour (pile[*B*−2] → I).

L'instruction de saut inconditionnel est JMP 0, **a** et **a** est simplement transféré dans le registre *I*. Les deux instructions de saut conditionnel, JPC **l**, **a** et JPP **l**, **a**, fonctionnent différemment selon que **l** est égal à 1 ou à 0. S'il est égal à 1, le saut à l'adresse **a** sera effectué si la valeur au sommet est différente de 0, s'il est égal à 0, le saut à l'adresse **a** sera effectué si la valeur au sommet est 0. La seule différence entre JPC et JPP est que JPC enlève le sommet contrairement à JPP.

L'instruction MSG permet de faire appel à un module externe écrit dans un autre langage, C par exemple. On peut voir cela comme un signal d'interruption sur un processeur Intel de la famille x86. Les valeurs de **l** et de **a** sont passées au code gérant le module d'appel externe et la procédure correspondante enregistrée pour ces valeurs est appelée. Elle aura accès aux valeurs dans la pile et à celles dans la mémoire des chaînes de caractères.

3.3 Bibliothèque de code

Une série d'objets prédéfinis composent l'environnement de CWS. La figure 24 décrit les liens père-fils entre les instances de ces objets, qui sont :

- Une fenêtre (**Window**).
- Des cartes (**Cards**).
- Une pile (**Stack**).
- Un bouton (**Button**).
- Un port d'entrée-sortie (**Port**).

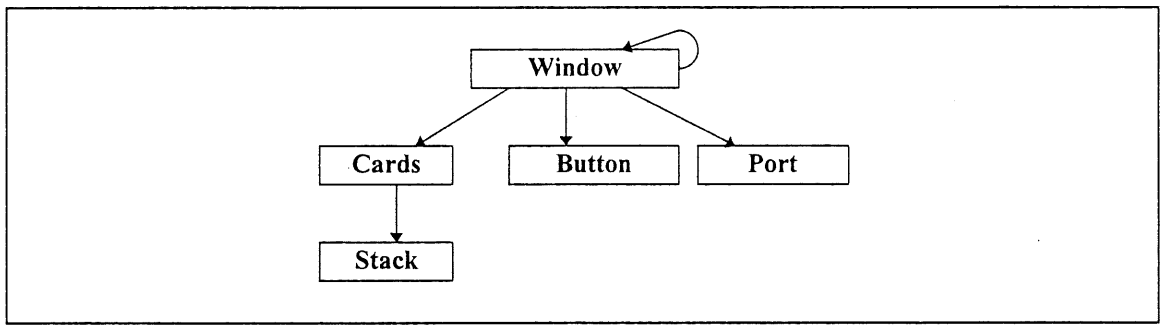


Figure 24 - Liens de parenté des objets prédéfinis

Ces objets sont décrits dans les sous-sections suivantes. Chacune est subdivisée ainsi :

- Méthodes d'instance, c'est-à-dire méthodes définies à l'intérieur de l'objet et redéfinissables par chaque instance.

- Méthodes d'objet, c'est-à-dire méthodes définies à l'extérieur de l'objet et non redéfinissables.
- Attributs constants, c'est-à-dire constantes définies à l'intérieur de l'objet et redéfinissables par chaque instance.
- Attributs variables, c'est-à-dire variables définies à l'intérieur de l'objet et redéfinissables par chaque instance.
- Instructions, c'est-à-dire instructions prédéfinies dans le langage qui ne s'appliquent qu'aux instances de ce type.

3.3.1 Fenêtre (Window)

Une instance de fenêtre peut en avoir une autre comme parent. Cet objet offre une surface quadrillée virtuellement à ses enfants : cartes (**Cards**), boutons (**Button**) et port d' entrée-sortie (**Port**). Selon la dimension physique, au moment de l'exécution, cet objet s'occupe de changer les dimensions physiques de ses enfants. Il a aussi une ligne d'information d'en-tête à laquelle du texte peut être ajouté. Une instance de base existe dans chaque programme, il s'agit de la fenêtre principale (**`MainWindow`**).

L'interaction entre l'utilisateur et le programme est gérée par du code qui répond aux messages de la souris et qui les transfère aux instances concernées. Ce code fait partie de la gestion interne de l'objet fenêtre. Il permet de définir ce qu'est une transaction. L'algorithme, simplifié pour un seul bouton de souris, est le suivant :

```
ATTENDRE message de la souris

TYPE DU message EST

    "bouton enfoncé" :
        CHERCHER instance à la position de la souris
        SI trouvée une pile (autre que Cursor)
            DÉBUTER transaction
            ENVOYER SelectFrom à l'instance
            SI il n'y a pas de cartes sur l'instance Cursor
                TERMINER transaction
                SI non Integrity? du système RECULER transaction
        SINON SI trouvé un bouton
            DÉBUTER transaction
            ENVOYER Select à l'instance
            TERMINER transaction
            SI non Integrity? du système RECULER transaction
        FIN

    "souris déplacée" :
        SI transaction commencée
            DEPLACER instance Cursor à la position de la souris
        FIN

    "bouton relaché" :
        SI transaction commencée
            CHERCHER instance à la position de la souris
            SI trouvée un pile (autre que Cursor)
                ENVOYER SelectTo à l'instance
                SI il n'y a pas de cartes sur l'instance Cursor
                    TERMINER transaction
                    SI non Integrity? du système RECULER transaction
                SINON
                    TERMINER transaction
                    RECULER transaction
            SINON
                TERMINER transaction
                RECULER transaction
        FIN
```

Méthode d'instance

- **`Init`** : elle est appelée une seule fois, à la construction de l'objet. Elle peut permettre d'initialiser certaines variables d'une instance. Par défaut, elle est vide.

Méthodes d'objet

- **`Show`** / **`Hide`** : elles permettent de rendre visible ou invisible une instance de fenêtre.
- **`Paint`** : permet de forcer la mise à jour visuelle d'une instance de fenêtre. suite par exemple à la modification du texte d'en-tête.
- **`Clear`** : permet d'effacer le texte d'en-tête de la fenêtre.
- **`WriteString`** / **`WriteInteger`** / **`WriteCard`** : permettent d'écrire dans l'en-tête. Le texte est ajouté au texte actuel. Elles permettent respectivement d'y ajouter une chaîne de caractères ou un entier ou une carte transformée en chaîne de caractères, ex. : 'ACE OF SPADE'.

Attributs constants

- **`parent`** : cette constante de type fenêtre permet d'attribuer un parent à l'objet. Par défaut, **`MainWindow`**.

- **X / Y / W / H** : de type entier permettent de définir la position de la fenêtre sur la surface de son parent. Ils définissent le coin supérieur droit (**X** et **Y**), la largeur (**W**) et la hauteur (**H**) en unité de la fenêtre parent (voir **MW** et **MH**).
- **MW / MH** : de type entier permettent de définir la dimension de la matrice offerte aux enfants. Ils deviennent les unités dans lesquelles les **X**, **Y**, **W** et **H** des enfants sont exprimés.

Attributs variables

- **text** : cette chaîne de caractères contient le texte d'en-tête de la fenêtre. Elle est normalement manipulée par les méthodes **Clear**, **WriteString**, **WriteInteger** et **WriteCard**.
- **modified** : cette variable booléenne indique si la fenêtre devrait être redessinée. Elle est utilisée par les méthodes **Clear**, **WriteString**, **WriteInteger** et **WriteCard**.

3.3.2 Cartes (Cards)

Une instance de cartes doit avoir comme parent une fenêtre. Elle est le père d'une série de piles (**Stack**) auxquelles elle offre de partager une collection d'images, comme les cartes d'un jeux de cartes ou les tuiles d'un jeu de tuiles tel MahJong. Elle s'occupe d'ajuster les dimensions physiques de ses enfants au moment de l'exécution et de calculer les

dimensions des images fournies pour qu'elles soient d'allure semblable. L'instance peut aussi se synchroniser avec un de ses frères lors du calcul des dimensions, une particularité intéressante. Supposons, par exemple, deux piles étalées de haut en bas et de gauche à droite. Dans la vraie vie, sur la pile étalée de haut en bas, si l'espace est restreint, la suite (pique, coeur, carreau ou trèfle) sera cachée par la carte sous-jacente. À cette fin deux instances prédéfinies de l'objet cartes, **`VCards`** et **`HCards`**, offrent des images de cartes avec la suite à droite ou en bas de la valeur. La synchronisation des deux types de cartes, **`VCards`** et **`HCards`**, est nécessaire pour que les cartes des deux piles aient la même dimension.

Les cartes normales dans une instance de l'objet cartes sont définies ainsi :

- 0..51 cartes normales de face
- 52..103 cartes normales retournées (le dos de la carte)
- 104..155 cartes normales de face mais grisées

Méthode d'instance

- **`Init`** : Elle est appelée une seule fois, à la construction de l'instance. Elle peut permettre d'initialiser certaines variables de l'instance. Par défaut elle est vide. Dans le cas des deux instances prédéfinies, cette méthode est re-définie pour créer les jeux de cartes voulus en utilisant l'instruction FACE.

Attributs constants

- **`parent`** : cette constante de type fenêtre permet d'attribuer un parent à l'objet. Par défaut **`MainWindow`**.
- **`sync`** : de type cartes permet de spécifier une autre instance de cartes sur laquelle on veut synchroniser les caractéristiques des cartes de l'instance en cours de définition.
- **`W / H`** : permettent de définir la dimension moyenne d'une image de carte sur la matrice du parent.
- **`RatioW / RatioH`** : permettent de définir un rapport largeur / hauteur à maintenir pour les images de l'instance cartes. Si ces valeurs sont 0, aucun rapport n'est maintenu.

3.3.2.1 Instruction

La figure 25 décrit la syntaxe en BNF de l'instruction FACE, la seule instruction prédéfinie de l'objet cartes. Cette instruction permet d'ajouter des images à la collection tenue par l'objet cartes. Ces images peuvent provenir du jeu normal de cartes (NORMAL) avec le texte horizontal ou vertical, d'une copie d'une autre image présente dans la collection (FACE) qu'on peut griser si on le désire (SIDE SHADED) ou d'un fichier externe contenant une image (BITMAP, où ***string*** correspond au nom du fichier).

statement ::=	FACE *card_interval* IS *card_statement*
card_statement ::=	NORMAL (HORIZONTAL | VERTICAL) |
	FACE *expression* [SIDE SHADED] |
	BITMAP *string* [SIDE SHADED]

Figure 25 - BNF de l'instruction de cartes

La figure 26 illustre des utilisations de l'instruction FACE au moyen de la définition d'une instance fictive **Piques** . Les treize premières images sont les cartes standard, de l'as de pique au roi de pique. Les treize suivantes sont la même suite de pique mais grisées, alors que les vingt-six dernières sont identiques et chargées à partir d'un fichier externe.

```
Cards Piques is
  W := 6;
  H := 6;
  Init is
    begin
    FACE 0..12 IS NORMAL HORIZONTAL;
    WITH I : integer DO
      FACE I IS FACE I - 13 SIDE SHADED
    FOR 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25;
    FACE 26..51 IS BITMAP 'dos_de_pique.bmp';
    end;
end Piques;
```

Figure 26 - Exemple de l'instruction FACE

3.3.3 Pile (Stack)

Une pile a comme parent une instance de cartes (**Cards**). Chaque instance peut avoir une direction d'empilement différente. Par exemple, les cartes peuvent être empilées de bas en haut, de gauche a droite, réparties horizontalement, etc. Chaque pile possède des méthodes répondant aux événements de souris sur sa surface : un retrait ou dépôt de cartes. Il y a quatre méthodes : dépôt ou retrait avec le bouton de gauche ou de droite et cinq méthodes multiples qui permettent de définir un comportement commun à plusieurs ou à tous les événements de souris possibles. Une méthode d'aide sera appelée si l'utilisateur interroge la pile. Une instance de pile prédéfinie existe pour le pointeur de souris (`Cursor`). Elle a la particularité de se déplacer en même temps que le pointeur de souris de l'utilisateur.

Méthodes d'instance

- `Start` : elle est appelée au début d'une partie. Elle peut, par exemple, être utilisée pour initialiser l'instance courante en allant chercher des cartes sur une autre (PULL). Le mécanisme de recul est mis en activité suite à l'appel de cette méthode, ce qui empêche de reculer au-delà de l'état initial du système.
- `SelectLeftFrom` / `SelectRightFrom` : sont utilisées pour répondre à une sélection avec le bouton gauche / droite de la souris sur la surface de la pile. Un paramètre de type INDEX indique sur quelle carte se trouve la souris.

- **SelectLeftTo / SelectRightTo** : sont utilisées pour répondre à un dépôt de cartes sur la surface de la pile avec le bouton gauche / droite de la souris. Un paramètre de type index indique sur quelle carte la souris se trouve. Ce paramètre peut valoir un de plus que la longueur de la pile si le dépôt est effectué à la fin de celle-ci.
- **Select** : cette méthode multiple sert à redéfinir les méthodes **SelectLeftFrom**, **SelectLeftTo**, **SelectRightFrom** et **SelectRightTo**;
- **SelectLeft** : sert à redéfinir les méthodes **SelectLeftFrom** et **SelectLeftTo**;
- **SelectRight** : sert à redéfinir les méthodes **SelectRightFrom** et **SelectRightTo**;
- **SelectFrom** : sert à redéfinir les méthodes **SelectLeftFrom** et **SelectRightFrom**;
- **SelectTo** : sert à redéfinir les méthodes **SelectLeftTo** et **SelectRightTo**;
- **Help** : elle est appelée lorsque le système est en mode d'aide et qu'un bouton de la souris est utilisé sur la surface de la pile. Elle n'a pas de valeur par défaut. Elle devrait être utilisée pour faire appel aux fonctions prédéfinies du sous-système d'aide.

Méthode d'objet

- **`Paint`** : elle permet de forcer la mise à jour visuelle de l'instance, suite par exemple à un ajout ou à une modification de cartes. Elle est appelée automatiquement suite à une transaction mais peut être appelée manuellement au milieu de celle-ci.

Attributs constants

- **`X / Y / W / H`** : ces constantes de type entier permettent de définir la position de la pile sur la surface de son parent. Elles définissent le coin supérieur droit (**`X`** et **`Y`**), la largeur (**`W`**) et la hauteur (**`H`**) en unités de la fenêtre parent (voir **`MW`** et **`MH`** de l'objet fenêtre).
- **`direction`** : permet de spécifier comment les cartes sont empilées : UP, DOWN, LEFT, RIGHT, HORIZONTAL, VERTICAL, LHORIZONTAL ou UVERTICAL. Les quatre premiers choix indiquent un empilement normal de bas en haut (UP), de haut en bas (DOWN), de droite à gauche (LEFT) et de gauche à droite (RIGHT). Les quatre derniers indiquent un empilement espacé régulièrement sur la surface de la pile horizontalement de droite à gauche (HORIZONTAL), horizontalement de gauche à droite (LHORIZONTAL), verticalement de haut en bas (VERTICAL) ou verticalement de bas en haut (UVERTICAL).
- **`handler`** : de type cartes spécifie quelle instance est le parent de la pile. Les images sur la pile proviendront de la collection d'images de ce parent.

3.3.3.1 Instructions

La figure 27 décrit les instructions de pile prédéfinies. Elles font référence à une ou deux piles et dans tous les cas ces références peuvent être présentes ou absentes. Quand une référence est absente, l'instruction est appliquée à l'objet contextuel. Ce qui veut dire que la procédure en cours de définition doit être une méthode d'objet ou une méthode d'instance de pile.

Figure 27, pour clarifier, on décrit ***stack_src*** et ***stack_dst*** comme des ***id***, en fait on devrait indiquer que n'importe quelle expression qui se réduit à une instance ou une référence de pile est acceptée. Un ***card_interval*** est une valeur de type CARD ou une suite numérique de valeurs de type CARD exprimée par les valeurs de début et de fin séparées par deux petits points comme en PASCAL. Ces valeurs référencent les éléments de la collection de l'instance cartes parent de la pile (**`handler`**). Un ***range*** est un indice de type INDEX ou une suite contiguë d'indices de type INDEX, exprimée par la valeur de début et celle de fin, dans la collection de cartes sur la pile.

Ces instructions sont décrites dans les sous-sections suivantes où, pour clarifier les explications, on considérera les références de piles existantes et les expressions ***card_interval*** et ***range*** comme une série d'éléments.

statement ::=	ADD *card_interval* [TO *stack_dst* [*range*]] |
	REMOVE [*stack_dst*] *range* |
	INVERSE [*stack_dst*] *range* |
	TURN [*stack_dst*] *range* SIDE (UP | DOWN | SHADED) |
	MOVE [*stack_src*] *range* TO [*stack_dst*] *range* |
	PULL *expression* [FROM *stack_src*] [TO *stack_dst*] |
	FLASH [*stack_dst*] *range* |
	INVERSE [*stack_dst*] *range* |
	SHUFFLE [*stack_dst*] |
stack_src ::= *id*	
stack_dst ::= *id*	
card_interval::=	*expression* ['..' *expression*]
range::=	'[' *expression* ['..' *expression*] ']'

Figure 27 - BNF des instructions de pile

ADD et REMOVE : ajout et retrait sur une pile

Elles permettent d'ajouter et d'enlever des éléments sur une pile. Les éléments ajoutés sont créés et ne proviennent donc pas d'une autre pile, alors que ceux enlevés sont détruits. Dans le cas d'un ajout on spécifie une série de cartes et une pile sur laquelle ajouter. Facultativement, on peut spécifier un point d'insertion dans la pile[15]; s'il est absent les cartes seront ajoutées à la fin. Dans le cas d'un retrait, on doit spécifier exactement l'emplacement de la suite de cartes à détruire. La figure 28 est un exemple de ADD et REMOVE montrant les éléments d'une instance de pile **A1** avant et après.

[15] dans ce cas ci, un seul INDEX est accepté

```
A1 → |1, 2, 3, 4, 5|
     ADD 6..9 TO A1[3];
A1 → |1, 2, 6, 7, 8, 9, 3, 4, 5|
     REMOVE A1[2..4];
A1 → |1, 8, 9, 3, 4, 5|
```

Figure 28 - Exemple de ADD et REMOVE

MOVE et PULL : déplacement d'une pile à l'autre

Elles permettent de transporter une série de cartes d'une pile à l'autre. Dans le cas de PULL un nombre donné de cartes est dépilé d'une pile source et empilé à une pile destination, sans en changer la séquence. Dans le cas de MOVE une série de cartes, n'importe où dans la pile, est transportée à un point spécifique d'une autre pile. Sémantiquement PULL est une MOVE avec :

```
PULL n FROM i1 TO i2

MOVE i1[i1! - n + 1 .. i1!] TO i2[i2! + 1]
```

La figure 29 donne un exemple de MOVE et PULL sur deux instances **A1** et **A2**.

```
A1 → |1, 8, 9, 3, 4, 5|
A2 → ||
      PULL 3 FROM A1 TO A2;
A1 → |1, 8, 9|
A2 → |3, 4, 5|
      MOVE A2[2..3] TO A1[3];
A1 → |1, 8, 4, 5, 9|
A2 → |3|
```

Figure 29 - Exemple de MOVE et PULL

INVERSE : changement d'ordre dans une pile

Elle permet d'inverser l'ordre d'une série d'éléments d'une pile. Elle est utile par exemple lorsqu'on veut ramener le contenu d'une pile de rebut dans le talon. La pile de rebut est habituellement le résultat du dépilage des éléments un par un de la pile de talon, donc son ordre est l'inverse de celui du talon. Si on ramène les cartes d'un seul PULL elles seront en ordre inverse. La figure 30 est un exemple d'INVERSE montrant les éléments d'une pile **A1** avant et après.

```
A1 → |1, 8, 9, 3, 4, 5|
      INVERSE A1[2..5];
A1 → |1, 4, 3, 9, 8, 5|
```

Figure 30 - Exemple d'INVERSE

TURN : changement de côté dans une pile

Elle permet de changer de côté une série de cartes. Comme décrit dans l'objet cartes, une carte peut être de face, de dos ou grisée. Si on considère UP, DOWN et SHADED comme étant ***side*** = 0, ***side*** = 1 et ***side*** = 2, appliquer TURN à un élément ***a*** consiste à :

```
a := (a MOD 52) + side * 52
```

La figure 31 donne une exemple de l'application de TURN sur une instance **A1**.

```
A1 → |1, 8, 9, 3, 4, 5|
     TURN A1[2..5] SIDE DOWN;
A1 → |1, 60, 61, 55, 56, 5|
     TURN A1[2..5] SIDE UP;
A1 → |1, 8, 9, 3, 4, 5|
     TURN A1[2..5] SIDE SHADED;
A1 → |1, 112, 113, 107, 108, 5|
```

Figure 31 - Exemple de TURN

FLASH : clignotement d'une pile

Elle permet d'attirer l'attention sur une pile en faisant clignoter une série de cartes. Les cartes sélectionnées clignotent trois fois. C'est une fonction cosmétique.

SUFFLE : battre les cartes d'une pile

Elle permet de battre les cartes d'une pile. L'algorithme est illustré figure 32. Après un certain temps, nous nous sommes aperçu qu'il avait un défaut : le hasard peut permettre à certaines cartes de rester à leur position originale. Nous voudrions changer cet algorithme pour celui figure 33 mais une des caractéristiques de l'environnement de jeu nous en empêche. Il s'agit du fait que l'utilisateur peut entrer le germe de départ (seed) pour la fonction de génération de nombre au hasard, ce qui permet de rejouer n'importe quelle partie en réutilisant le même germe. Changer l'algorithme empêcherait de ré-utiliser des germes intéressants trouvés dans une version précédente du logiciel.

```
FOR longueur de la pile * 2 TIMES
        Echange deux éléments au hasard dans la pile
```

Figure 32 - Premier algorithme de battage

```
FOREACH  élément de la pile
        Echange avec un autre élément au hasard dans la pile
```

Figure 33 - Deuxième algorithme

3.3.4 Bouton (Button)

Un bouton doit avoir comme parent une fenêtre (**Window**). Il répond à une pression de l'utilisateur avec la souris sur sa surface. Un texte peut y apparaître, ce qui permet de donner une indication sur l'effet qu'aura l'utilisation du bouton. Une méthode spéciale

Update sera exécutée après chaque transaction avec la souris. Cela permet de mettre à jour le texte du bouton en fonction de l'état courant du système suite à chaque transaction.

Méthodes d'instance

- **Init** : est appelée une seule fois, à la construction de l'instance. Elle peut permettre d'initialiser certaines variables de l'instance ou d'écrire un texte définitif sur le bouton. Par défaut elle est vide.
- **Update** : appelée après chaque transaction, permet de changer le texte ou l'état du bouton (visible / invisible) en fonction de l'état courant du système.
- **Select** : appelée lorsqu'un bouton de la souris est appuyé sur sa surface.

Méthodes d'objet

- **Show / Hide** : permettent de rendre un bouton visible ou invisible.
- **Paint** : permet de forcer la mise à jour visuelle d'un bouton, suite par exemple à la modification de son texte.
- **Clear**: permet d'effacer le texte du bouton.
- **WriteString / WriteInteger / WriteCard** : permettent d'écrire sur le bouton. Le texte est ajouté a son contenu. Elles permettent respectivement d'y ajouter

une chaîne de caractères ou un entier ou une carte transformée en chaîne de caractères, ex. : 'ACE OF SPADE'.

Attributs constants

- **`X` / `Y` / `W` / `H`** : ces constantes de type entier permettent de définir la position du bouton sur la surface de son parent. Elles définissent le coin supérieur droit (**`X`** et **`Y`**), la largeur (**`W`**) et la hauteur (**`H`**) en unités de la fenêtre parent (voir **`MW`** et **`MH`** de l'objet fenêtre).
- **`parent`** : de type fenêtre permet d'attribuer un parent à l'objet. Par défaut **`MainWindow`**.

Attribut variable

- **`text`** : cette chaîne de caractères contient le texte du bouton. Elle est normalement manipulée par les méthodes **`Clear`**, **`WriteString`**, **`WriteInteger`** et **`WriteCard`**.

3.3.5 Port d'entrée-sortie (Port)

Un port doit avoir comme parent une fenêtre (**Window**). Il permet d'écrire et de lire du texte sur l'écran. Comme pour l'objet bouton (**Button**), une méthode spéciale **`Update`** sera exécutée après chaque transaction avec la souris. Ceci permet de mettre à jour le

texte du port en fonction de l'état courant du système suite à la transaction. Une fonction **`Read`** permet d'aller chercher le texte entré par l'utilisateur.

Méthodes d'instance

- **`Init`** : est appelée une seule fois, à la construction de l'instance. Elle permet d'initialiser certaines variables de l'instance ou d'écrire un texte définitif sur un port de sortie ou un texte de départ sur un port d'entrée. Par défaut elle est vide.
- **`Update`** : appelée après chaque transaction, permet de changer le texte ou l'état du port (visible / invisible) en fonction de l'état courant du système.

Méthodes d'objet

- **`Show`** / **`Hide`** : permettent de rendre un port visible ou invisible.
- **`Paint`** : permet de forcer la mise à jour visuelle d'une instance de l'objet port, suite par exemple à la modification de son texte.
- **`Clear`**: permet d'effacer le texte du port.
- **`WriteString`** / **`WriteInteger`** / **`WriteCard`** : permettent d'écrire sur le port. Le texte est ajouté à son contenu. Elles permettent respectivement d'y ajouter une chaîne de caractères ou un entier ou une carte transformée en chaîne de caractères, ex. : 'ACE OF SPADE'.

- **Read** : permet d'aller chercher le texte actuel du port. Une valeur de type STRING est retournée. On utilise cette fonction plutôt que d'accéder directement à la variable **text** car elle vérifie, dans le cas d'un port d'entrée, si de nouveaux caractères se sont ajoutés depuis la dernière transaction.

Attributs constants

- **X / Y / W / H** : ces constantes de type entier permettent de définir la position du port sur la surface de son parent. Elles définissent le coin supérieur droit (**X** et **Y**), la largeur (**W**) et la hauteur (**H**) en unités de la fenêtre parent (voir **MW** et **MH** de l'objet fenêtre).
- **parent** : de type fenêtre permet d'attribuer un parent à l'objet. Par défaut **MainWindow**.
- **style** : une combinaison de valeurs permet de définir le style du bouton : avec barre de défilement horizontale (**ps_hscroll**) et/ou verticale (**ps_vscroll**) ou non (défaut), port d'entrée (**ps_edit**) ou de sortie (défaut), plusieurs lignes (**ps_multiline**) ou une seule (défaut) et alignement à gauche (**ps_left**), à droite (**ps_right**) ou au centre (**ps_center**).

Attribut variable

- **text** : cette chaîne de caractères contient le texte du port. Elle est normalement manipulée par les méthodes **Clear**, **WriteString**, **WriteInteger**, **WriteCard** et **Read**.

CONCLUSION

Dans l'état présent, développer un programme CDL depuis ses spécifications jusqu'au produit fini, le programme exécutable et un module d'explication des règles, prend de quinze à soixante-quinze minutes. Un générateur de programmes CDL de jeux de tuiles à partir d'un dessin en caractères ASCII a récemment été ajouté à l'environnement. Un second programme, plus générique, pour composer graphiquement des programmes CDL est en développement. Il permet de créer des instances, les placer sur une surface matricielle et définir leurs comportements. Une interface à la bibliothèque de codes et prototypes prédéfinis y est intégrée. Ces deux ajouts vont permettre de réduire le temps de création.

Un objet **Brain** a été créé et ajouté à l'environnement CDL + CWS dans le cadre d'un projet d'intelligence artificielle et un jeu de backgammon autodidacte a été implanté. Suite à des résultats intéressants, un développement dans ce domaine est prévu.

Deux autres projets qui nous intéressent dans un proche avenir sont l'ajout d'héritage simple lors de la définition de catégories de prototypes et celui de fonctionnalités dynamiques comme la création de groupes et d'instances. Nous voulons aussi étudier l'impact des changements dynamiques sur le comportement des instances.

Notre tendance au niveau des modifications à CDL est d'essayer de sortir les constructions spécifiques aux objets **Stack** et **Cards** (instructions, tableaux implicites,

etc.) pour rendre le langage générique. Nous cherchons une formule pour pouvoir décrire des instructions à partir du fichier source, ce qui, encore une fois, nous permettrait de garder la compatibilité antérieure.

En attendant qu'il nous rende riche, nous allons continuer de développer CWS et d'ajouter d'autres succès et déboires à sa genèse.

ANNEXES

Annexe A - Terminologie des jeux de solitaires

Valeur	Comme dans la plupart des jeux de cartes, as, deux, trois, etc. , incluant les faces, valet, dame et roi.
Suite	Pique, coeur, carreau ou trèfle.
Couleur	Rouge ou noir.
Le Tableau	Consiste en une seule carte, un groupe de cartes ou une pile de cartes ayant des utilités et limites différentes selon les jeux.
Fondations	Sont des cartes sur lesquelles d'autres cartes sont jouées pour bâtir des séquences complètes, avec pour but de compléter la partie. Les fondations peuvent faire partie du tableau original ou peuvent être définies pendant le jeu, différemment selon les jeux.
Le Talon	S'applique au restant du paquet après la distribution du tableau.
Le Rebut	Consiste en des cartes qui ne peuvent être jouées une fois distribuées et qui doivent être mises de côtés. Certains jeux se terminent lorsque le talon à été transféré dans la pile de rebut. D'autres permettent de réutiliser la pile de rebut comme nouveau talon.
La Réserve	Est un paquet ou groupe de cartes qui est mis de côté ou retenu pour jouer sur les fondations.
Séquence Ascendante	Suite commençant par une carte basse, habituellement un As, et montant jusqu'à une carte haute, i.e. A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K.
Séquence Descendante	Suite commençant pas une carte haute, habituellement un Roi, et descendant jusqu'à une carte basse, i.e. K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, A.

Cartes Auxiliaires	Cartes appartenant au tableau, pouvant être jouées sur les fondations ou être utilisées pour former des séquences temporaires, en fonction des règles du jeu.
Cartes en Lignes	Cartes distribuées horizontalement sur le tableau, de façon unique ou avec recouvrement, selon le jeu.
Cartes en Colonnes	Cartes distribuées verticalement sur le tableau, de façon unique ou avec recouvrement, selon le jeu.
Cartes Disponibles	Cartes libres pour bâtir sur les fondations ou pour transférer sur les cartes ou colonnes auxiliaires.
Cartes Bloquées	Cartes devant être libérées d'une certaine façon pour devenir Disponibles.

BIBLIOGRAPHIE

[Aho89] Aho, Alfred / Sethi, Ravi / Ullman, Jeffrey, COMPILATEURS Principes, techniques et outils, InterEditions, Paris, 1989

[Coi87] Cointe, Pierre, *Metaclasses are First Class : the ObjVlisp model*, OOPSLA 1987, Special Issue of SIGPLAN Notices, vol 22, no 12, pp. 156-167, Orlando, Floride, USA, octobre 1987

[Coo39] Coops, Helen L., 100 Games of Solitaire, Whitman publishing company, Racine, Wisconsin, U.S.A., 1939

[Dav96] Davis, Stephen R., Learn Java Now, Microsoft Press, Redmond, Washington, 1996

[Mas89] Masini, Gérard / Napoli, Amedeo / Colnet, Dominique / Léonard, Daniel / Tombre, Karl, Les langages à objets, InterEditions, Paris, France, 1989

[Mey88] Meyer, Bertrand, Object-Oriented Software Construction, Prentice Hall, 1988.

[Wir75] Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1975